Antônio Sérgio Lima Aragão

ASPECTOS TÉCNICOS E METODOLÓGICOS DO PROCESSO DE DESENVOLVIMENTO DE APLICAÇÕES BASEADO EM COMPONENTES

Dissertação apresentada ao Programa de
Pós-Graduação em Engenharia de Produção da
Universidade Federal de Santa Catarina
como requisito parcial para obtenção do grau de
Mestre em Engenharia de Produção.

Orientador: Prof. Fernando Álvaro Ostuni Gauthier, Dr.

Florianópolis

2002

Ficha Catalográfica

A659a Aragão, Antônio Sérgio Lima
Aspectos técnicos e metodológicos do processo de desenvolvimento de aplicações baseado em componentes. – Florianópolis: 2002.
151p.

1. Informática – Monografia.
I. Título

CDD 001.5

Antônio Sérgio Lima Aragão

ASPECTOS TÉCNICOS E METODOLÓGICOS DO PROCESSO DE DESENVOLVIMENTO DE APLICAÇÕES BASEADO EM COMPONENTES

Esta dissertação foi julgada e aprovada para a obtenção do grau de **Mestre em Engenharia de Produção** no **Programa de Pós-Graduação em Engenharia de Produção** da Universidade Federal de Santa Catarina.

Florianópolis, ___ de _____ de 2002.

_____

Prof. Ricardo Miranda Barcia, PhD.

Coordenador do Programa

BANCA EXAMINADORA

_____

Prof. Fernando Álvaro Gauthier, Dr.

Universidade Federal de Santa Catarina

Orientador

_____

Profa. Edis Mafra Lapolli, Dra.

Universidade Federal de Santa Catarina

_____

Prof. Alejandro Martins, Dr.

Universidade Federal de Santa Catarina

Agradecimentos

Primeiramente, quero agradecer à minha amada esposa Luzia Helena e a meus queridos filhos Thaíssa e Filipe, pela compreensão, companhia e apoio neste tempo que dediquei ao mestrado. Sei que foram momentos de sacrifícios e que não se recuperam mais. Muitas vezes me proporcionaram a tranqüilidade e o incentivo necessários para que pudesse me aprofundar nas pesquisas. Espero poder proporcionar bons momentos proximamente. Sou eternamente grato a vocês.

Quero agradecer, com grande amor e respeito, aos meus pais José e Conceição (em memória) que não mediram esforços para me possibilitar uma boa formação acadêmica e que me foi muito valioso durante toda a minha vida profissional. Agradeço principalmente os seus exemplos de fé, perseverança e de dedicação que me transmitiram.

Agradecimentos aos colegas da DATAPREV de Brasília, que me incentivaram na caminhada em busca deste título. Grato, também, a DATAPREV que me proporcionou a liberação de parte do período de trabalho, durante a realização do mestrado. Espero poder retribuir a confiança com muito trabalho.

Agradeço também aos meus alunos do curso Superior em Tecnologia de Processamento de Dados da UNEB, Rogério e Mary, que contribuíram com este trabalho através do desenvolvimento do estudo de caso. Foi muito bom orientá-los no seu projeto final de graduação, onde desenvolveram o protótipo da camada de persistência utilizando a linguagem Java.

Um agradecimento especial ao professor Gauthier, que mesmo à distância, soube orientar e direcionar os meus estudos para que pudesse escrever da melhor forma possível este trabalho.

“As novas tecnologias, sozinhas, não teriam mudado o mundo se os organizadores não lhes tivessem aproveitado oportunamente a potencialidade, não as tivessem introduzido oportunamente nos sistemas produtivos, nas burocracias e nos serviços, não tivessem preparado as estratégias, os homens, os procedimentos, os locais adequados para exaltar as vantagens e reduzir os perigos.”

Domenico de Masi

Resumo

ARAGÃO, Antônio Sérgio Lima. **Aspectos técnicos e metodológicos do processo de desenvolvimento de aplicações baseado em componentes.** 2002. 153f. Dissertação (Mestrado em Engenharia de Produção). Programa de Pós-Graduação em Engenharia de Produção, UFSC, Florianópolis.

Componentes de software oferecem um paradigma de desenvolvimento de software onde uma aplicação final será montada por meio da conexão desses componentes, independentes de linguagem e plataforma, adquiridos de terceiros ou desenvolvidos pelo próprio usuário. Com esse paradigma se pretende conseguir os benefícios de um desenvolvimento mais rápido de aplicações com um grau maior de qualidade e segurança e um custo menor do que o desenvolvimento tradicional de aplicações.

Para possibilitar que as organizações adotem o desenvolvimento de aplicações baseado na utilização de componentes é preciso adotar uma metodologia de desenvolvimento padronizada, já utilizada com sucesso no mercado, promover a capacitação dos seus técnicos nessa nova tecnologia, bem como promover uma mudança organizacional que permita que as informações referentes a aspectos funcionais e não-funcionais dos componentes estejam disponíveis de maneira padrão, para que sejam possíveis uma busca e utilização mais eficientes.

O objetivo principal deste trabalho é analisar os aspectos técnicos e gerenciais de algumas metodologias de desenvolvimento de aplicações e as dificuldades operacionais de implantação dessa tecnologia nas organizações. As metodologias RUP - Rational Unified Process, CATALYSIS e VINCIT foram selecionadas por já contarem com alguns anos de utilização.

A utilização de uma das metodologias analisadas foi feita através do estudo de caso do desenvolvimento de um componente de uma Camada de Persistência para utilização de um banco de dados relacional por uma aplicação escrita em Java.
Palavras-chave: componente de software, metodologia de desenvolvimento, orientação a objetos, camada de persistência, UML, CBD.

Abstract

ARAGÃO, Antônio Sérgio Lima. **Aspectos técnicos e metodológicos do processo de desenvolvimento de aplicações baseado em componentes.** 2002. 153f. Dissertação (Mestrado em Engenharia de Produção). Programa de Pós-Graduação em Engenharia de Produção, UFSC, Florianópolis.

Software components offer a paradigm of software development where a final application will be set up through the connection of those components, independent of language and platform, acquired of third or developed by the own user. With that paradigm it intends to get the benefits of a faster development of applications with a larger degree of quality and safety and a smaller cost than the traditional applications development.

To make possible that the organizations adopt the development of applications based on the use of components it's necessary to adopt a development methodology standardized, already used with success in the market, to promote your technicians' training in that new technology, as well as to promote a change organizations that allows that the referring information to functional and no-functional aspects of the components they are available in a standard way, so that it is possible a search and more efficient use.

The objective principal of this work is to analyze the technical and management aspects of some methodologies of development of applications and the operational difficulties of implantation of that technology in the organizations. The methodologies RUP - Rational Unified Process, CATALYSIS and VINCIT were selected for they already count with some years of use.

The use of one of the analyzed methodologies was made through the study of case of the development of a component of a Layer of Persistence for use of a relational database for an application written in Java.

Keywords: software component, development methodology, object oriented, persistence layer, UML, CBD.

SUMÁRIO

Lista de Figuras

Lista de Quadros

Lista de Siglas

ANSI	American National Standardization
API	Application Program Interface
CA	Computer Associates, Inc.
CASE	Computer Aided Software Engineering
CBD	Component-Based Development
CBSE	Component-Based Software Engineering
CMM	Capability Maturity Model
CNPJ	Cadastro Nacional de Pessoa Jurídica
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CVRD	Companhia do Vale do Rio Doce
DBA	Database Administrator
DCOM	Distributed Component Object Model
E-Commerce	Comércio Eletrônico
ECOOP	European Conference on Object-Oriented Programming
EJB	Enterprise JavaBeans
E-Mail	Endereço eletrônico
ERP	Enterprise Resource Provider
HP	Hewlett Packard, Inc.
HTML	HiperText Markup Language
IDL	Interface Definition Language
IEEE	Institute of Eletronic and Eletric Engineer
JDBC	Java Database Conectivity
JVM	Java Virtual Machine
MDS	Metodologia de Desenvolvimento de Sistemas
OD	Objetos Distribuídos
ODBC	Open Database Conectivity
OID	Object Identifier
OMG	Object Management Group
OO	Orientação a Objetos
RMI	Remote Method Invocation
RUP	Rational Unified Process

SEI	Software Engineering Institute
SQL	Structured Query Language
UML	Unified Modeling Language
USP	Universidade de São Paulo
Web	Relativo à Internet

1 INTRODUÇÃO

1.1 Motivação

O crescimento acentuado do uso da informática nas organizações, o surgimento cada vez mais acelerado de novas tecnologias, o aumento da complexidade dos aplicativos e o maior número de pessoas envolvidas fizeram com que o mercado de produção de sistemas se tornasse cada vez mais competitivo.

As empresas têm centrado as suas necessidades na capacitação de seus empregados, como requisito essencial para ganhar produtividade, com qualidade. No entanto, as empresas vêm enfrentando problemas estruturais como a necessidade da redução de pessoal para a diminuição dos custos operacionais, aliados aos problemas técnicos advindos das mudanças tecnológicas.

Belloquim (jan/97, p.10), no artigo “Migração Tecnológica É um Processo Cultural”, salienta que “é fácil verificar que o investimento em tecnologia, quando entendido como compra de hardware e software, não tem produzido os resultados esperados em aumento de qualidade, produtividade e satisfação das necessidades dos usuários que se prometia”. O autor afirma que “a complexidade inerente às novas tecnologias tem deixado perdidos e sem mapa os desenvolvedores e gerentes de desenvolvimento”.

Como poderemos adquirir o conhecimento sobre as novas tecnologias, introduzí-las no ambiente de desenvolvimento, através de treinamentos dirigidos e agregar valor aos aplicativos produzidos com qualidade e produtividade? Belloquim(jan/97, p.10-11), chama a atenção para o fato de que:

> “As novas tecnologias de produção de sistemas têm se mostrado muito mais complexas do que o esperado no seu lançamento, tanto pela dificuldade inerente de se adquirir e empregar essas tecnologias, como pela diversidade existente, dificultando a escolha do padrão a ser seguido. Além disso, a maioria dos desenvolvedores, após longos anos trabalhando em aplicações desenvolvidas com base em técnicas de Análise Estruturada, tem grandes dificuldades (quando não resistência explícita) de passar, de uma hora para outra, para um ambiente com dados e processos encapsulados em objetos, interfaces gráficas, objetos distribuídos, ferramentas CASE, e-mail e arquiteturas em três camadas. Além disso, muitas tecnologias recentes não estão ainda maduras o suficiente, para o desenvolvimento de aplicações de missão corporativa”.

Nesse sentido é de fundamental importância a conscientização dos gerentes das áreas de informática de que a migração para uma nova tecnologia, como

arquitetura em n camadas, sistemas para Internet ou *e-commerce*, é muito mais uma questão de mudança cultural, que uma mudança meramente tecnológica, que deve começar pelos técnicos mais experientes, pois estes têm os recursos necessários para identificar os riscos técnicos.

1.2 Justificativa

Segundo Tindall (2000) o desenvolvimento de um bom sistema corporativo para a Internet é uma das tarefas mais árduas da área de informática. Esse tipo de aplicação é bem complexo por vários motivos, principalmente pela abrangência das tecnologias envolvidas e o número de requisitos necessários para o seu funcionamento ideal neste ambiente.

O desenvolvimento corporativo de sistemas é definido por Tindall (2000, p.5) como a "capacidade de suportar múltiplos locais, geografias, organizações e usuários com suas necessidades informacionais embutidas nos processos de negócios necessários para executar atividades essenciais de uma organização". Em outro trecho, Tindall (2000) ressalta que os sistemas corporativos de hoje devem ter a capacidade de interagir com serviços de mensagem, devem ser capazes de gerar páginas HTML dinamicamente, acessar as mais diversas bases de dados, algumas vezes fazer transações com *mainframes* ou outros serviços legados.

A performance de um sistema com *interface web,* como qualquer outro sistema informatizado, é crucial e por isto se deve atentar ao compartilhamento de recursos, já que o número de usuários de um sistema internet pode ser centenas de vezes maiores que uma aplicação *stand alone* ou em uma rede corporativa. O controle de transações, a utilização de um servidor seguro para proteger o tráfego das informações e a utilização de criptografia para guardar dados confidenciais são outros fatores que não são simples de serem tratados.

Além destes problemas, a disponibilidade da aplicação é outro fator de relevância. Se o número de usuários de uma aplicação *web* é bem maior que de um sistema intranet, no entanto, pode crescer de forma absurda em poucos dias. Ainda segundo Tindall (2000), não são raros os casos de serviços que ficaram indisponíveis após a divulgação na mídia, como um comercial no intervalo da novela ou o envio de uma mala direta para possíveis clientes. Todos estes fatores tornam a tarefa de desenvolver sistemas algo extremamente complexo.

Décadas atrás, as pessoas utilizavam a linguagem *Assembler* para programar computadores. Com o tempo, chegaram as linguagens de alto nível que permitiram expressar as idéias de forma simples e compacta. Quando os programadores descobriram que eles perdiam muito tempo reprogramando estruturas de dados existentes, as bibliotecas de funções apareceram. Isto aumentou o nível de abstração e de reuso no desenvolvimento de aplicativos.

Depois chegou a Orientação a Objetos. As noções de encapsulamento, herança e polimorfismo, que serão explicitadas no próximo capítulo, provaram ser irresistíveis. Mas apenas programar com objetos, não garante o reuso. Foi apenas depois que a noção de objetos foi extendida às bibliotecas de classes, que o reuso começou a tomar forma. Várias bibliotecas comerciais de classes começaram a aparecer e a popularidade destas bibliotecas levou as pessoas a darem atenção na promessa realizada do reuso de software.

Ao mesmo tempo, o nível de abstração oferecido para o programador de aplicações, permitiu a programação de todos os objetos num nível onde, bibliotecas genéricas e específicas do domínio, forneciam lógicas prontas para o uso.

O esquema de utilização de lógicas prontas, no entanto foi logo mostrando seus problemas. Um deles, segundo Sant'Anna (fev/2001), é que as funções não estavam formalmente agrupadas para permitir o uso adequado e que por causa disso surgiram bibliotecas de classes. Bibliotecas de classes forneciam conjuntos de classes relacionadas que podiam ser utilizadas como estavam ou podiam ser especializadas, via herança, para resolver um problema.

Ainda segundo Sant'Anna (fev/2001, p.34-35), "... ferramentas como o *Visual Basic* e o *Delphi* introduziram um modelo completamente diferente: o usuário manipulava métodos, propriedades e eventos de componentes, que por sua vez se responsabilizavam por chamar a API do Windows", facilitando a interação da linguagem de programação com o sistema operacional.

Muitos problemas ainda podem ser observados em relação à utilização desta técnica, segundo o mesmo autor (Sant'Anna, fev/2001):

- Devem ser suportadas diversas linguagens de programação;
- Vários conceitos como propriedades e eventos devem ser suportados nativamente;
- Os objetos devem incluir informações detalhadas dos tipos incluídos para facilitar a chamada das classes e a validação do uso e manutenção do sistema;

- Criar objetos e até mesmo herdar uma classe da outra deve ser permitido mesmo que só tenhamos o código binário disponível e não saibamos a linguagem de desenvolvimento original;
- O gerenciamento de memória deve ser feito pelo sistema operacional, para que o programa possa "passar um objeto" para outro programa sem precisar se preocupar com a alocação de memória; e
- Deve existir uma preocupação com o controle de versões para garantir que programas possam utilizar diversas versões de uma mesma classe, conforme a sua necessidade.

A proposta deste trabalho é analisar as novas metodologias de desenvolvimento de sistemas que dão ênfase à utilização de componentes de *software*, possibilitando o reuso adequado desses componentes, e escolher uma dessas metodologias para desenvolver um projeto de um componente.

1.3 Objetivos

1.3.1 Objetivo geral

É objetivo geral desta pesquisa analisar os aspectos técnicos do desenvolvimento de aplicações baseado na tecnologia de componentes, bem como analisar três das mais conhecidas metodologias de desenvolvimento de sistemas orientadas a componentes existentes atualmente, focando no ganho de produtividade e qualidade que poderá ser alcançado pelas equipes de desenvolvedores.

Pretendemos, neste trabalho, obter uma visão clara e objetiva de como o processo de desenvolvimento de aplicações poderá ser melhorado, através da utilização da tecnologia de componentes, que usa extensivamente a herança, princípio básico da orientação a objetos, permitindo que os desenvolvedores trabalhem em soluções modulares, em níveis mais altos de abstração, com maior foco no negócio e maior produtividade.

1.3.2 Objetivos específicos

Para a consecução do objetivo geral, estão propostos os seguintes objetivos específicos:

- Prospectar as principais metodologias de desenvolvimento de sistemas que permitam a construção de aplicativos baseada em componentes;
- Analisar os aspectos técnicos e procedimentais de três metodologias existentes no mercado e com boa aceitação pelas empresas;
- Proceder uma avaliação dessas metodologias com a finalidade de verificar a sua utilização em equipes de desenvolvimento de software com diferentes graus de experiência;
- Elaborar um estudo de caso com a finalidade de utilizar uma das metodologias descritas anteriormente e conhecer, de forma prática, a construção de um componente que poderá ser utilizado em diversas aplicações;
- Identificar as principais dificuldades de implantação da tecnologia de componentes numa organização, que não esteja no estado da arte em termos tecnológicos, e as mudanças organizacionais imprescindíveis para a sua operacionalização.

1.4 Estrutura do trabalho

A metodologia empregada neste trabalho foi inicialmente a pesquisa em bibliografias específicas sobre o assunto e em *sites* da Internet, principalmente de Universidades. Foram escolhidas três metodologias mais significativas no mercado nacional e internacional, que incluíssem a tecnologia de produção de software baseada em componentes, para aprofundamento e apresentação destes estudos.

Desta forma, o capítulo 2 traz um levantamento sobre a tecnologia de objetos que serviu de base para o surgimento do desenvolvimento baseado em componentes. Descrevemos o que os autores entendem por componente e os benefícios que se espera da utilização de componentes. Na continuação descrevemos de maneira sucinta a tecnologia de objetos distribuídos e as suas principais tecnologias de apoio. Descrevemos também o que se entende por uma metodologia de desenvolvimento de sistemas e o que é a notação UML.

No capítulo 3 descrevemos as metodologias selecionadas conforme os seus autores, mostrando as suas principais características.

No capítulo 4 realizamos a modelagem e o projeto de um componente que foi escrito na linguagem Java, destinado a criar uma camada de persistência que

permita a utilização de um banco de dados relacional, a partir de aplicações orientadas a objeto.

O quinto capítulo conclui a dissertação mostrando uma análise das dificuldades em se implantar uma tecnologia baseada nas metodologias expostas e traz também sugestões para trabalhos futuros.

Concluindo, a bibliografia consultada é referenciada no final do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

2.1Introdução

Antes de aprofundar em definições teóricas é conveniente estabelecer o nível de detalhamento que pretendemos utilizar em nosso trabalho. Sabemos que as transformações tecnológicas que estão sendo introduzidas na indústria de software são muito grandes. Novas ferramentas e tecnologias estão emergindo em ritmo acelerado. Vários fornecedores de produtos têm apresentado soluções completas para o desenvolvimento de aplicativos que garantem ser a melhor solução.

Pretendemos focar inicialmente o trabalho nos principais conceitos da tecnologia de objetos que serviu de base para o surgimento do desenvolvimento baseado em componentes e o que se espera da utilização de componentes.

Na continuação, focaremos de forma sucinta os conceitos de objetos distribuídos com os principais suportes tecnológicos existentes, quais sejam COM/DCOM da Microsoft, CORBA (*Common Object Request Broker Architecture*) e EJB *(Enterprise Java Beans*). Esta visão de implementação de objetos distribuídos poderá ser tema de estudos futuros.

2.2 Orientação a objetos

A Orientação a Objetos (OO) é um conceito que surgiu no início da década de 80, sendo que a primeira linguagem comercial baseada em OO foi desenvolvida pelo Bell Labs em 1985. Na época, dominavam as linguagens estruturadas, como Pascal, C e Cobol. A qualidade do software dependia quase que exclusivamente da habilidade do programador. Os softwares eram de difícil gerenciamento e raramente os códigos eram aproveitados em outros programas. Atualmente, predominam as linguagens OO, como C++, Java, Delphi e J++, comprovando a superioridade desse novo conceito sobre os demais.

“Orientação a objeto pode ser definida como as disciplinas de modelagem de software que tornam fácil construir sistemas complexos a partir de componentes individuais” ou que “a orientação a objeto proporciona representação direta dos objetos físicos manipulados pelo usuário final e um paradigma de mensagem-objeto mais natural, para interagir com objetos” segundo Khoshafian (1994, p.6-7).

> “...uma aplicação no universo de objetos consiste de um conjunto de blocos de construção autocontidos e predefinidos que podem ser localizados, reparados ou substituídos. A

> construção de código autocontido propicia o teste completo antes de ser utilizado dentro da lógica do sistema de informação" de acordo com Furlan (1998, p. 16).

Uma grande vantagem oferecida pela OO é a possibilidade de reutilização de código. O programador usa o código já existente para criar seu próprio software, eliminando a necessidade de realizar uma mesma tarefa duplamente. O principal resultado é a economia no tempo de maturação do projeto. Além disso, a manutenção do software é facilitada. Se um módulo defeituoso é aproveitado por vários programas, basta corrigí-lo uma única vez e todos os programas terão acesso ao módulo já consertado. Nas linguagens estruturadas, para corrigir defeitos, o programador é obrigado a considerar os efeitos da propagação de erros no seu código. Essa tarefa pode ser impossível, dependendo da complexidade do software.

Em OO, é possível representar os objetos do mundo real, através de classes hierárquicas. Por exemplo, podemos representar pessoa como uma classe. A partir dela, uma divisão seria a subclasse pessoa física e uma outra, a subclasse pessoa jurídica. Essa representação hierárquica aumenta a organização do software, pois as características comuns podem ser agrupadas em uma mesma classe, ou seja, o esforço é concentrado em um único ponto. No exemplo anterior, toda pessoa possui um nome, não importando se é uma empresa ou não. Por outro lado, as características específicas de cada classe podem ser alteradas sem atingir o todo, encapsulando as mudanças sem afetar as outras classes. No exemplo, somente empresas possuem CNPJ.

Para ambientes corporativos, utilizando tecnologia OO, um projeto será implementado em menor tempo representando uma redução drástica nos custos, e também será aumentada a confiabilidade do resultado final, de acordo com Tindall (2000).

A realidade da informática hoje é centrada em Internet, em interfaces gráficas, em sistemas distribuídos, em comércio eletrônico e já fazem parte do dia-a-dia de grandes corporações em todo o mundo. Para desenvolver os novos sistemas para esta realidade, a tecnologia de orientação a objetos é um ponto chave. Uma equipe de desenvolvimento utilizará ferramentas de desenvolvimento de software disponíveis no mercado, otimizadas para o uso com a orientação a objetos. Os principais bancos de dados dos grandes fabricantes já são categorizados como objeto-relacionais.

Sérgio Mainetti Jr., em entrevista que consta no *site* da Visionnaire, empresa especializada em soluções orientadas a objeto, diz que:

> "basta uma simples navegação pela Internet para confirmar o forte uso da orientação a objetos como tecnologia atual: (1) ferramentas CASE gerando código automaticamente a partir de modelos de objetos, (2) ambientes de desenvolvimento visuais, poderosos e robustos, levando maior produtividade para o desenvolvedor, (3) applets Java (que nada mais são do que objetos) distribuídos pela Internet aos milhões, se comunicando, e alcançando uma alta taxa de reuso, (4) sistemas complexos executando, e que não requerem impossíveis planos de reestrutura da arquitetura interna por causa de manutenções que são e sempre serão necessárias".

Como toda tecnologia que amadurece comprova seu benefício e passa a ser utilizada por todos, a orientação a objetos já está se transformando em algo transparente (tanto para o desenvolvedor como para o usuário). Orientação a objetos está em todos os lugares, desde o *kernel* (núcleo central) dos sistemas operacionais mais modernos, passando por infra-estruturas completas voltadas para o desenvolvimento de software, e chegando ao nível da aplicação voltada para negócios (até mesmo um analista financeiro que precisa criar macros em uma planilha eletrônica está trabalhando com orientação a objetos).

"A maturidade e as vantagens desta tecnologia já estão comprovadas, os próximos avanços trarão tecnologias derivadas que fornecerão ainda mais vantagens e produtividade para o desenvolvimento de software" segundo Mainetti Jr na mesma entrevista.

2.2.1 Classes e objetos

De acordo com Furlan (1998) define-se um objeto como uma entidade tangível que exibe certos comportamentos definidos. Em termos formais, um objeto é uma unidade, item ou entidade do espaço-tempo, real ou abstrata, individual e identificável, que modela uma parte da realidade. Um objeto possui **estado**, **comportamento** e **identidade**. O comportamento e a estrutura de objetos similares são definidos em uma classe comum.

2.2.1.1 O estado de um objeto

O estado de um objeto compreende todas as suas propriedades (em geral, estáticas) e os correntes valores das mesmas (em geral, dinâmicos), segundo Furlan

(1998). Uma propriedade é uma característica herdada ou distinta que contribui para que um dado objeto se torne único. As propriedades são, em geral, estáticas, já que os atributos são comumente imutáveis e fundamentais à natureza do objeto. O fato de todo objeto possuir estado implica que todo objeto ocupa espaço, ou no mundo real ou em memória.

2.2.1.2 O comportamento de um objeto

Define-se comportamento como sendo a forma pela qual um objeto age e reage, em termos de mudanças de estado e de passagem de mensagens, segundo Furlan(1998). O comportamento de um objeto, portanto, é completamente definido por suas ações.

Uma operação é uma ação que um objeto executa sobre outro de forma a produzir uma reação.

Todos os métodos associados a uma determinada classe compõem o **protocolo** dos objetos daquela classe, definindo o comportamento permissível a tais objetos, e englobando a visão externa que se deve ter dos objetos.

A existência de estado em um objeto implica que a ordem na qual as operações são aplicadas é importante, o que leva à idéia de objetos como “máquinas” independentes. Assim, para cada objeto, a ordem das operações no tempo é tão importante que pode-se formalizar a caracterização do comportamento de um objeto em termos de uma máquina de estados finitos equivalente.

2.2.1.3 A identidade de um objeto

A identidade é a propriedade de um objeto que o distingue de todos os demais, também segundo Furlan (1998). A maioria das linguagens de programação usa nomes de variáveis para distinguir objetos temporários, combinando identidade e endereçamento. A maioria dos bancos de dados, por sua vez, utiliza-se de chaves identificadoras para distinguir objetos persistentes, combinando identidade e valores de dados.

Devido à sua distinção dos demais, a identidade de cada objeto é preservada até mesmo quando seu estado muda por completo.

2.2.1.4 A fábrica de objetos

Uma classe pode ser analisada sob dois pontos de vista: uma **visão externa**, que é providenciada pela **interface** da classe, (declarações das operações aplicáveis aos objetos da classe, podendo incluir declarações de outras classes, variáveis, etc.) enfatizando a abstração e escondendo sua estrutura e seu comportamento; ou a partir de uma **visão interna,** que é garantida pela **implementação** de seu comportamento, ou seja, das operações pré-definidas na interface da classe, de acordo com Ambler (1999).

Segundo Ambler (1999), a *interface* de uma classe pode ser dividida em diversas partes:

- **pública**: uma declaração que é parte da *interface* de uma classe e é visível em qualquer parte do ambiente de referenciamento da classe;
- **protegida**: uma declaração que é parte da interface de uma classe mas não é visível a quaisquer outras classes, exceto às suas subclasses;
- **privada**: uma declaração que é parte da interface de uma classe, indicando que não é visível a nenhuma outra classe, inclusive a suas subclasses.

O estado de um objeto é representado por declarações de constantes / variáveis colocadas na parte privada da interface de uma classe. Assim, a representação comum dos objetos de uma classe é encapsulada, e mudanças em sua representação não afetam nenhum de seus “clientes”.

2.2.1.5 Relacionamentos entre classes

Há três tipos básicos de relacionamentos entre classes de objetos, de acordo com Furlan (1998):

- **generalização** - que indica relacionamentos subclasse-superclasse;
- **agregação** - usada para denotar relacionamentos todo-parte, e
- **associação** - utilizada para denotar relacionamentos entre classes não correlatas.

2.2.1.6 Herança

Herança pode ser classificada como o relacionamento entre classes onde uma classe compartilha a estrutura e o comportamento já definidos em uma (herança simples) ou várias (herança múltipla) classes. A herança define uma

hierarquia entre classes, na qual uma **subclasse** herda de uma ou mais **superclasses,** segundo Ambler (1999).

A partir de herança, pode-se conceber a implementação de classes sem instâncias - **classes abstratas** -, as quais são idealizadas para serem moldes para eventuais subclasses, que irão adicionar sua estrutura e comportamento, geralmente sobrepondo métodos, segundo Ambler (1999). A classe mais geral em uma estrutura de classes é denominada classe base. A maioria das aplicações tem muitas classes que podem, a certo nível, ser consideradas como “base”, representando a mais generalizada categoria de abstração dentro de dado domínio do problema. Uma subclasse pode aumentar a estrutura de sua superclasse, mas nunca diminuí-la. Métodos podem ser sobrepostos, mas a declaração em uma subclasse de uma nova variável-membro com nome idêntico ao de uma variável-membro de sua superclasse é, em certos casos, encarada como erro.

2.2.1.7 Polimorfismo

Polimorfismo é um conceito de teoria de tipos no qual um nome pode denotar objetos de diferentes classes que são relacionados por alguma superclasse (ou, em último caso, por uma classe base) em comum, segundo ainda Ambler (1999).. Polimorfismo trata de funções de mesmo nome (desde que pelo menos um de seus parâmetros perfaça uma distinção) sejam invocadas para objetos diferentes. O polimorfismo é, portanto, bastante útil quando há muitas classes com o mesmo “protocolo”: cada objeto implicitamente “saberá” qual função ele acessará. Herança sem polimorfismo é possível, mas não muito satisfatória.

2.3 Componentes

"Um componente pode ser definido como um elemento reusável de software que é a menor unidade de distribuição e gerenciamento de software em tempo de projeto e execução. Os componentes são análogos a ”circuitos integrados” de software e representam um estilo de programação que está num nível de abstração muito alto”, segundo Braga (1998, p. 18-21).

“Um pacote coerente de software que pode ser desenvolvido independentemente e pode ser entregue como uma unidade, com uma interface definida pela qual pode ser composto com outros componentes, para prover e usar seus serviços” segundo o método Catalysis de D´Souza(1999, p.386).

Brown (2000, p. 75) descreve um componente como sendo “um pedaço independentemente utilizável de funcionalidade que provê acesso a seus serviços através de interfaces”. E diz mais “um componente é muito mais que uma sub-rotina em um programa modular, que um objeto ou uma classe na orientação a objetos ou um pacote em um modelo de sistema”.

Ainda segundo Brown (2000), um componente tem algumas características importantes, a saber:

- Um componente tem características de um pacote executável de software;
- Ele provê alguma funcionalidade para satisfazer alguma necessidade;
- Oferece serviços através de interfaces.

“Um componente é uma unidade de software independente em nível de aplicativo, desenvolvida para um propósito específico e não para um aplicativo específico” de acordo com Morisseau-Leroy (2001, p.23).

Para Booch (2000) um componente é um pacote coerente de artefatos de software que pode ser desenvolvido independentemente, podem ser entregues como uma unidade, tem explícito e bem-definida a sua interface para os serviços que provê, tem explicito também a interface para os serviços que espera de outros componentes, podendo ser composto, inalterado, com outros componentes para construir algo maior.

Uma estratégia de componente para desenvolvimento de software é baseada em construções fundamentais de paradigmas orientados a objetos. Morisseau-Leroy (2001) diz que embora os termos “componente” e “objeto” sejam freqüentemente usados indistintamente, um componente não é um objeto. Um objeto é uma instância de uma classe, enquanto que um componente pode ser uma classe, mas normalmente é uma coleção de classes e interfaces. Durante a execução, um componente se torna “vivo” quando suas classes são instanciadas.

Os componentes são, na maioria das vezes, objetos comerciais que têm comportamentos predefinidos e reutilizáveis. Para se utilizar um componente precisamos conhecer a sua interface com o mundo exterior, mas não precisamos conhecer os detalhes da sua implementação interna, que ficam ocultos nas interfaces, encapsulados em conjuntos de funcionalidades. As interfaces são o meio pelo qual os componentes se conectam e são constituídas por conjuntos de operações nomeadas, que são ativadas pelos clientes.

Segundo Morisseau-Leroy (2001, p.24) na implementação de sistemas baseados em componentes, “os provedores e clientes se comunicam através da especificação da interface, que se torna a camada intermediária, permitindo que as duas partes colaborem e trabalhem juntas”.

[1][2][3]Szyperski (1998) citando a ECOOP (1996) que um componente de software é uma unidade de composição com interfaces especificadas por contrato e apenas com dependências de contexto explícitas. Depois diz que um componente de software pode ser distribuído independentemente e está sujeito à composição por terceiros, ou seja, componentes de diversos fabricantes podem ser utilizados numa mesma aplicação, interagindo entre si.

 que usar componentes para o desenvolvimento de aplicações?

Os componentes elevam o nível de abstração da resolução do problema de tal forma que se pode utilizá-los, sendo um especialista no domínio, sem precisar ser um programador experiente. A utilização de componentes permite que os fornecedores construam ambientes de desenvolvimento visuais nos quais o conceito de conectar estes componentes de software forme a base de qualquer novo desenvolvimento.

A necessidade de codificação escrita é realmente mínima - roteiros podem ser usados para colar os componentes ou para ajustar funcionalidades existentes. Um projeto típico realizado com componentes consiste apenas em importar os componentes de interesse e personalizá-los sem codificação explícita e, finalmente, uní-los para formar uma aplicação, segundo Brown(2000). Essa nova tecnologia de desenvolvimento de sistemas foi chamada de Desenvolvimento Baseado em Componentes (*Component-Based Development* – CBD) ou ainda Engenharia de Software Baseada em Componentes (*Component-Based Software Engineering* – CBSE).

Para os desenvolvedores e usuários de sistemas, o CBD é um caminho para reduzir os custos de desenvolvimento, aumentar a produtividade e permitir a atualização controlada dos sistemas face à rápida evolução tecnológica.

As vantagens da utilização de componentes de software são múltiplas, segundo Morisseau-Leroy (2001):

- Independente – Um componente é mais generalizado e é independente do aplicativo.

- Reutilizável – Os componentes são bens reutilizáveis. Comparados com soluções específicas para problemas específicos, se obtém maior produtividade pelo reuso do modelo e da implementação.
- Qualidade - Aumento da confiabilidade pelo uso de código bem testado.
- Montagem - Vários componentes podem ser montados em paralelo para formar um sistema funcional.
- Custos - Menores custos de manutenção em conseqüência de uma base de código menor.
- Fácil de atualizar e manter – A atualização de componentes individuais acaba com o problema de atualizações em massa, como acontece com os sistemas monolíticos.
- Localização transparente - Os componentes fornecem um mecanismo bem encapsulado de empacotar, distribuir e reusar software, podendo estar em qualquer lugar de uma rede – em computadores mais convenientes para executá-los.
- Distribuído – Usando-se padrões de sistemas de computação distribuída, como *Enterprise JavaBeans* – EJB, *Common Object Request Broker Architecture* - CORBA ou *Component Object Model / Distributed Component Object Model* - COM / DCOM da Microsoft, os componentes de software podem ser distribuídos por toda uma rede corporativa.

2.4 Objetos distribuídos

As redes corporativas de grandes organizações são caracterizadas pela heterogeneidade. Os sistemas heterogêneos consistem em uma combinação de numerosos sistemas operacionais, espalhados por múltiplos componentes de hardware e de software. A construção de uma infra-estrutura de software capaz de permitir a comunicação entre os esses diversos componentes se constitui num novo paradigma da computação. O conceito mais recente desenvolvido em sistemas de computação distribuída é o de objetos distribuídos.

Segundo Morisseau-Leroy (2001) a computação de objetos distribuídos se refere a aplicativos que fazem chamadas de ativação remota a outros programas residentes em diferentes computadores e/ou diferentes redes, oferecendo suporte ao intercâmbio e à reutilização de objetos distribuídos e permitindo aos

desenvolvedores construir sistemas a partir da montagem de componentes de diferentes fabricantes.

Existem vários paradigmas de objetos distribuídos:

- Para normatizar o mercado de objetos distribuídos(OD), foi criada em 1989 uma associação mundial, a OMG – *Object Management Group* – que congrega mais de 800 empresas de porte: IBM, Compaq, HP, British Telecom, Sun, Inprise/Borland, Microsoft, Netscape, Oracle, Informix, CA, Nortel, Baan, Fujitsu, entre outras. Os usuários também estão presentes nessa associação. Podem ser citados entre eles: Boeing, CNN, Citibank, AT&T, Sprint, Alcatel, British Telecom, Matsushita. No Brasil, Hospital das Clínicas de SP, USP, Visionnaire, Unicamp, entre outros. Entre alguns usuários de OD no país estão os Correios, Embrapa, Michelin, HSBC, BankBoston, Banco do Brasil, Xerox, Siemens, Telepar, Copel, Itaipu e a CVRD. A OMG existe para definir os padrões. Uma das principais contribuições da organização até o momento é o padrão CORBA que define como os objetos devem operar em um ambiente distribuído. Os objetos e interfaces CORBA são especificados usando-se a OMG IDL – *Interface Definition Language*. A OMG IDL permite a operação mútua entre objetos no lado do cliente e no lado do servidor, escritos em diferentes linguagens de programação e distribuídos em diversas plataformas de hardware.
- O DCOM (*Distributed Component Object Model*) , desenvolvido pela Microsoft, é uma tecnologia de componente para distribuição de aplicativos na arquitetura Windows. Os objetos e interfaces COM são especificados usandos-se a IDL (*Interface Definition Language*) da Microsoft.
- O RMI (*Remote Method Invocation*) Java da Sun JavaSoft permite que um objeto Java sendo executado em uma JVM (Java Virtual Machine – máquina virtual Java) chame métodos presentes em outro objeto Java sendo executado em outra JVM.
- A arquitetura de componentes JavaBeans da Sun JavaSoft permite que os desenvolvedores criem componentes no lado do cliente que podem ser montados usando-se construtores de aplicativos visuais e dispositivos de janela não-visuais.

- O EJB(Enterprise JavaBeans) é um modelo de componentes que permite aos desenvolvedores distribuírem componentes nos servidores de aplicativos e servidores de banco de dados. Em aplicativos EJB, a ativação remota segue a especificação RMI, mas os fornecedores não estão limitados ao protocolo de transporte RMI.

2.5 Metodologia de desenvolvimento de sistemas

2.5.1 Introdução

As empresas modernas têm descoberto que o gerenciamento da informação como um recurso é uma questão estratégica para a sobrevivência em mercados cada vez mais competitivos e frente a fontes de recursos cada vez mais escassas.

Diante dessa situação, os sistemas de informação tornaram-se peças-chave no gerenciamento da informação corporativa. O sucesso no desenvolvimento, operação e manutenção tem significado para as empresas, flexibilidade e capacidade competitiva no mercado, permitindo mais precisão e agilidade na tomada de decisão. Como já comentado no Capítulo 1, os sistemas atuais tendem, cada vez mais, a serem mais complexos, maiores e mais dinâmicos do que os sistemas concebidos anteriormente, necessitando portanto, serem mais organizados, versáteis e flexíveis.

Uma das grandes preocupações das organizações é a necessidade de se criar sistemas de informações com qualidade, produtividade e baixo custo. Por isto, tornou-se imprescindível a adoção de métodos que possam proporcionar mais estabilidade ao processo de desenvolvimento de sistemas de informação.

Os métodos convencionais de desenvolvimento de sistemas (análise e projeto estruturado) apresentam algumas limitações que introduzem pontos de descontinuidade no processo de desenvolvimento de sistemas de informação. Estas limitações ocorrem principalmente por serem utilizados diagramas diferentes em cada fase do ciclo de vida de um sistema, acarretando perda de informação e inconsistências na transição de um diagrama para outro, conforme citado por Furlan (1998).

Os métodos orientados a objetos utilizam um modelo único, o qual é utilizado em todas as fases do ciclo de vida de um sistema. Dessa forma, o processo de desenvolvimento de sistemas é simplificado, interativo e controlável. Cada iteração

acrescenta características ao modelo, de forma a haver menores possibilidades de inconsistências e erros, conforme descrito por Furlan (1998).

O desenvolvimento de sistemas baseado em objetos concentra a maior parte do esforço na fase de análise de requisitos. Esse esforço adicional é compensado pela implementação mais rápida e mais simples do projeto. O sistema resultante é mais completo e de fácil manutenção devido ao encapsulamento e ao reuso.

Várias metodologias orientadas a objeto surgiram no mercado, algumas com grande utilização e outras com pouca divulgação, como as seguintes que tiveram alguma importância:

Quadro 1: Metodologias orientadas a objeto.

Sigla	Ano	Título	Autores
RDD	1990	Responsibility-Driven Design	R. Wirfs-Brock
OOD	1991	Object - Oriented Design	G. Booch
OMT	1991	Object Modeling Technique	J. Raumbaugh
OOA/OOD	1991	Object - Oriented Analysis / Object - Oriented Design	P. Coad / E. Yourdon
OOSE	1992	Object - Oriented Software Engineering	I. Jacobson
VMT	1994	Visual Modeling Tecnique	IBM
FUSION	1994	Fusion	D. Coleman
OOAD	1994	Análise e Projeto Orientados a Objeto	J. Odell / J. Martin
Objectory	1998	Objectory	G. Booch / J. Raumbaugh / I. Jacobson

Fonte: Furlan (1998, p.24-29)

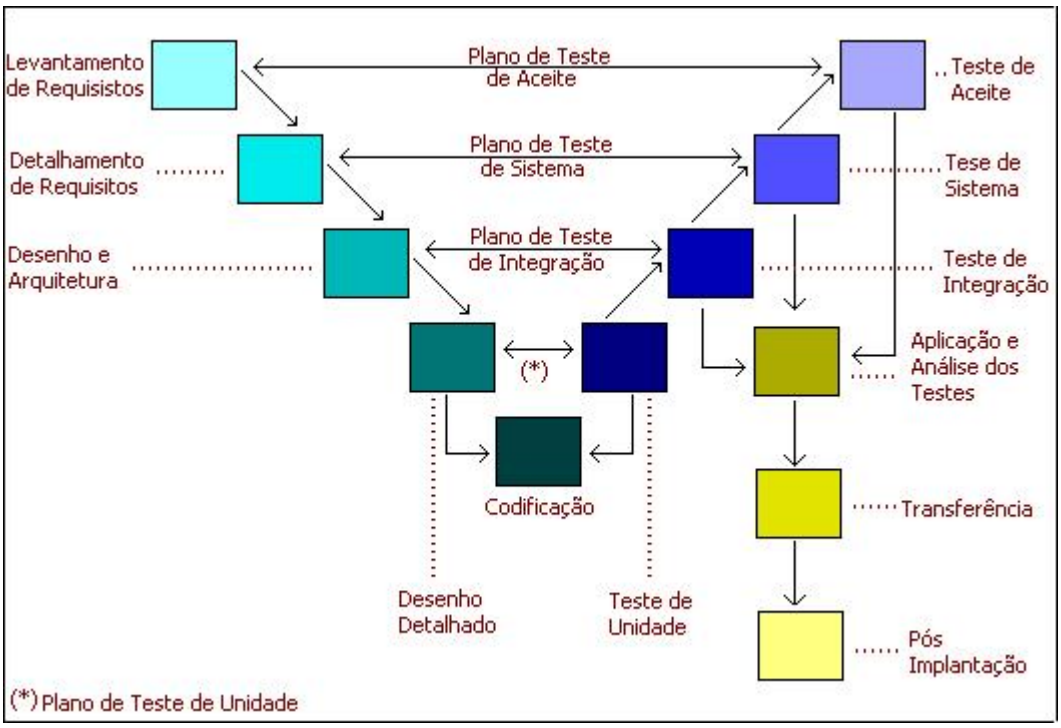
Assim como os softwares e hardwares, os métodos orientados a objetos têm evoluído para subsidiar as novas tecnologias de desenvolvimento disponíveis no mercado.

2.5.2 Engenharia de software

Uma metodologia de desenvolvimento de sistemas é um *framework* de processos organizacionais que, se usados adequadamente, garantem o sucesso da área de Engenharia de Software.

O conceito de Engenharia de Software, tomando como base o modelo proposto pela ESA (obtido através dos padrões relevantes ANSI/IEEE), envolve as fases apresentadas na figura 1, segundo Sakamoto(2000):

Figura 1: Fases para a Engenharia de Software



Fonte: Sakamoto (2000)

2.5.2.1 Levantamento de requisitos

Corresponde à fase inicial de levantamento das necessidades e desejos do usuário, dividida nas sub-fases de pré-proposta (ou estudo de viabilidade) e após a proposta aprovada. Com a definição dos requisitos do projeto são também definidos os seus critérios de aceite, provendo clareza na aplicação dos Testes de Aceite finais.

2.5.2.2 Detalhamento de requisitos

Corresponde ao detalhamento técnico dos requisitos levantados na fase anterior. Com o detalhamento de requisitos acontece também a definição de um Plano de Teste de Sistema.

2.5.2.3 Desenho da arquitetura

Nesta fase é elaborado o Plano de Testes de Integração e também a definição da arquitetura do projeto de software, principalmente no que se refere aos componentes, aos dados, às interfaces e o ambiente de execução.

2.5.2.4 Desenho detalhado

Fase do detalhamento das especificações técnicas de implementação (codificação) dos componentes de software e da especificação dos Testes de Unidade.

2.5.2.5 Codificação

Fase de implementação dos componentes de software definidos e detalhados nas fases anteriores. São trabalhados os aspectos relativos à construção dos programas propriamente ditos.

2.5.2.6 Testes

Fase de aplicação dos planos de teste, elaborados em fases anteriores e o resultado é submetido à avaliação para tratamento das eventuais não-conformidades encontradas.

2.5.2.7 Transferência

Esta fase envolve os procedimentos necessários para a transferência do ambiente de desenvolvimento para o de produção, acompanhamento de implantação e/ou geração do setup para o cliente final.

2.5.2.8 Pós-implantação

Contempla a fase de pós implantação das aplicações, coleta de lições aprendidas, melhores práticas, satisfação do usuário final (inclusive via eventuais Solicitações de Alteração) e estimativas realizadas.

Além dessas fases, a Engenharia de Software contempla as técnicas de modelagem para a geração dos produtos (modelos) necessários para o sucesso de cada etapa. As técnicas de modelagem mais comumente usadas são a UML – para sistemas orientados a objetos; a Análise Essencial – sistemas orientados a eventos; e a Modelagem de Dados – quando a armazenagem de dados é feita em uma Base de Dados Relacional. A orientação de uso das técnicas, produtos gerados, papéis e responsabilidade de cada fase é formalizada com a geração de uma metodologia de desenvolvimento de software (MDS).

Outro aspecto relevante na Engenharia de Software é a definição de métricas. A medição coerente e assistida ao longo do ciclo de vida do desenvolvimento de software permite a coleta e mensuração da performance dos processos e produtos gerados, dando subsídios para o estabelecimento de uma melhoria contínua no ambiente de desenvolvimento de software e na definição de estimativas com maior grau de precisão.

Para garantir que as fases da engenharia sejam executadas de forma eficiente, os produtos saiam dentro de um padrão de qualidade esperado, os prazos e as estimativas estabelecidas sejam cumpridas e haja o comprometimento entre a área solicitante e a executora, é fundamental que haja uma disciplina de gerenciamento.

Com o objetivo de dar suporte aos itens citados anteriormente, agilizando a geração dos produtos de cada fase, surgem as ferramentas de apoio. Neste contexto são usadas ferramentas para gerenciamento de configuração, gerenciamento de requisitos, ferramenta CASE e de Teste, objetivando suportar o processo de Engenharia de Software.

2.5.3 Evolução para os métodos baseados em componentes

Segundo D´Souza (1998), quando projetamos soluções baseadas em componentes, não precisamos saber como eles são representados, se como objetos ou como instâncias de uma classe ou saber como os conectores entre componentes trabalham. Da mesma maneira que em programação OO, objetos têm que ter acesso à informação armazenada por outros objetos, assim em uma arquitetura baseada em componentes, estes se intercomunicam por interfaces bem definidas, para preservar o encapsulamento mútuo.

Os componentes de pequeno tamanho são bem parecidos com as classes de objetos. Na realidade eles seriam mais fáceis de implementar como classes de objetos em uma linguagem de programação orientada a objeto. Isto mostra que as diferenças entre projeto baseado em componentes e projeto orientado a objeto são principalmente de tamanho e escala e não são intrínsecos de outro tipo de projeto.

As principais características da tecnologia de componentes segundo D'Souza (1998) são:

- Componentes usam freqüentemente armazenamento persistente; embora objetos em uma linguagem de programação OO tenham sempre um estado local,

eles trabalham tipicamente dentro da memória principal, e a persistência é negociada separadamente.

- Componentes têm uma maior gama de mecanismos de intercomunicação, como eventos e *workflows*, em lugar de só a mensagem básica da orientação a objetos. Estes mecanismos apóiam a composição mais fácil das suas partes.
- Componentes são freqüentemente de maior tamanho do que os objetos tradicionais e podem ser implementados como múltiplos objetos de classes diferentes. Eles têm, freqüentemente, ações mais complexas em suas interfaces, em lugar de únicas mensagens de objetos.
- Um pacote de componentes, por definição, inclui definições das interfaces que são disponibilizadas como também das interfaces que requer; os únicos focos de definição das classes tradicionais são as operações providas e não as operações requeridas.

Componentes, como objetos, interagem por interfaces de polimorfismo. Todas as técnicas de modelagem se aplicam igualmente bem em ambos os casos, inclusive os conectores mais gerais para componentes. Mas, podemos freqüentemente falar sobre um exemplo de componente, um tipo de componente, e classe de componente. Também podem ser construídos componentes sem usar técnicas de projeto orientado a objeto; mas a utilização da OO torna a modelagem bem mais fácil.

D`Souza (1998) comenta ainda que não deveríamos assumir que objetos são intrínsecos ao desenvolvimento baseado em componentes; na realidade, uma das vantagens de componentes é que eles podem encapsular sistemas legados (sistemas provenientes de arquiteturas antigas que são implementados dentro de componentes).

Outra característica da abordagem orientada a objetos é que ela também colabora para aumentar a capacidade adaptativa dos sistemas de negócio das organizações. É o benefício oriundo da reusabilidade, ou seja, o reaproveitamento de código, viabilizado pela aplicação dos seus conceitos de herança e da modelagem, que visa alta coesão e baixo acoplamento, que tem permitido a criação de componentes de *software* reutilizáveis, segundo Larman (2000).

3 ALTERNATIVAS METODOLÓGICAS

3.1 Introdução

As metodologias foram selecionadas em função de sua penetração no mercado, da existência e disponibilidade de material para pesquisa (livros, cd-rom ou na internet), em função do seu tempo de vida (todas elas são metodologias com três ou mais anos de lançamento) e também por serem já conhecidas no Brasil.

Uma outra característica que também procuramos adotar como parâmetro de seleção foi o fato de utilizarem a notação UML – *Unified Modeling Language* -padrão mundial estabelecido pela OMG. Todas as metodologias utilizam a UML para a elaboração dos seus modelos.

As metodologias selecionadas são as seguintes:

- RUP – Rational Unified Process
- CATALYSIS
- VINCIT

Essas metodologias envolvem todas as fases do ciclo de desenvolvimento de sistemas, inclusive o gerenciamento do processo.

3.2 Metodologia RUP – Rational Unified Process

3.2.1 Introdução

A Rational Software é uma empresa especializada no oferecimento de soluções para desenvolvimento e implantação de software, conforme informações constantes no *site* da empresa. Foi constituída por três das maiores autoridades em orientação a objetos que são Grady Booch, James Rumbaugh e Ivar Jacobson.

A Metodologia RUP - Rational Unified Process, também chamada de Processo Iterativo Controlado, tem como proposta apoiar e ajudar seus usuários a desenvolver e implantar software com mais qualidade e de maneira mais eficiente. A Rational concentra as suas atividades na aplicação daquilo que considera serem as seis mais importantes práticas para desenvolvimento e implantação de software:

- Gerência de Requisitos
- Modelagem Visual
- Implementação em Componentes
- Teste Automatizado
- Gerência de Configuração e Versão

- Processo Iterativo Controlado (RUP – *Rational Unified Process*)

Para cada uma das técnicas acima, a empresa possui produtos de software e serviços que facilitam a condução da implementação dessas técnicas.

A descrição do método RUP foi baseada nos livros de Philippe Kruchten (*The Rational Unified Process – An Introduction Second Edition* 2000), no livro de G. Booch, J. Rumbaught e de I. Jacobson (UML Guia do Usuário 2000, p. 442-448), que são os autores do método RUP, e no livro de José Davi Furlan (1998, p.239-248).

3.2.2 Características do processo

O RUP é um processo iterativo que proporciona uma compreensão crescente do problema por meio de aperfeiçoamentos sucessivos e do desenvolvimento incremental de uma solução efetiva em vários ciclos. Este enfoque permite a flexibilidade para acomodação de novos requisitos ou de mudanças táticas de objetivos de negócio, bem como que sejam identificados e solucionados os riscos inerentes ao projeto desde o início dos trabalhos.

As atividades do RUP dão ênfase à criação e manutenção de modelos, principalmente aqueles especificados com a utilização da UML, que proporcionam representações que podem ser visualizadas de várias maneiras e onde as informações podem ser capturadas e controladas através de uma ferramenta de apoio. Os modelos do RUP são baseados nos conceitos de objetos e classes e nos relacionamentos existentes entre eles e utilizam a UML como notação comum.

O *Rational Unified Process* proporciona um desenvolvimento centrado na arquitetura. O processo focaliza o desenvolvimento inicial e a linha de base da arquitetura de um software, proporcionando o desenvolvimento paralelo, minimizando a necessidade de refazer o trabalho e aumentando a probabilidade de reutilização de componentes e melhorando a capacidade de manutenção eventual do sistema. Este projeto de arquitetura serve como base para o planejamento e o desenvolvimento de software a partir de componentes.

As atividades desenvolvidas sob a orientação do RUP são orientadas por casos de uso. As noções de casos de uso e cenários são empregadas para alinhar o fluxo do processo a partir da captura dos requisitos por meio de testes e para

proporcionar controles que poderão ser acompanhados ao longo do desenvolvimento do sistema.

O Rational Unified Process é um processo configurável, podendo ser ajustado e redimensionado para atender às necessidades de projeto que variam desde pequenas equipes até grandes empresas de desenvolvimento de software. Existe uma orientação para se configurar o processo e atender às necessidades de uma empresa.

O controle de qualidade e o gerenciamento de riscos, contínuos e objetivos, são inseridos no processo, em todas as atividades e envolvendo todos os participantes, de forma que os riscos para o sucesso do projeto são identificados e atacados no inicio do processo de desenvolvimento, quando há tempo suficiente para uma reação adequada.

3.2.3 Fases e iterações

Uma fase é o período de tempo entre dois importantes marcos de progresso do processo em que um conjunto bem-definido de objetivos é alcançado, artefatos são concluídos e decisões são tomadas em relação à passagem para a fase seguinte.

Conforme mostra a Figura 2, o *Rational Unified Process* é composto pelas seguintes quatro fases:

Quadro 2. Fases do RUP – *Rational Unified Process*

1. Concepção	Estabelece o caso de negócio para o projeto.
2. Elaboração	Estabelece um plano de projeto e uma arquitetura sólida
3. Construção	Desenvolve o sistema.
4. Transição	Fornece o sistema a seus usuários finais.

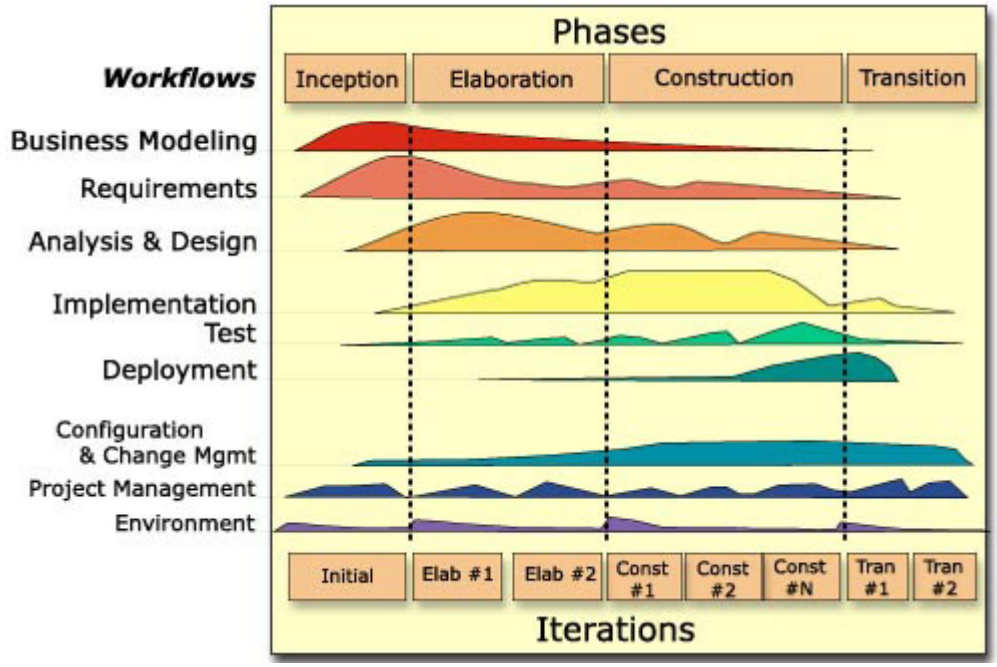
Fonte: Kruchten (2000)

A concepção e a elaboração abrangem as atividades de engenharia do ciclo de vida do desenvolvimento, enquanto que a construção e a transição constituem sua produção.

Em cada fase ocorrem várias iterações. Uma iteração representa um ciclo completo de desenvolvimento, desde a captação de requisitos na análise até a implementação e a realização de testes, resultando na versão de um projeto executável.

Cada fase e iteração têm algum foco de redução de riscos e concluem um marco de progresso bem-definido. A consideração do marco de progresso proporciona um ponto no tempo para avaliar como as metas foram alcançadas e se o projeto necessitará ser reestruturado de alguma maneira.para prosseguir.

Figura 3.1. O Ciclo de Vida de Desenvolvimento de um Software.



Fonte: Kruchten (2000, p.46)

3.2.3.1 Fases

3.2.3.1.1 Concepção

Durante a fase de concepção, você estabelece o caso de negócio para o sistema e delimita o escopo do projeto. O caso de negócio inclui critérios de sucesso, a avaliação de riscos, a estimativa de recursos necessários e um plano para a fase, mostrando a programação dos principais marcos de progresso. Durante a concepção, é comum a criação de um protótipo executável, servindo como teste para a concepção.

No final da fase de concepção, você examina os objetivos do ciclo de vida do projeto e decide se deve prosseguir com o desenvolvimento em plena escala.

3.2.3.1.2 Elaboração

As metas da fase de elaboração são a análise do domínio do problema, o estabelecimento da fundação de uma arquitetura sólida, o desenvolvimento do plano

do projeto e a eliminação dos elementos de mais alto risco do projeto. As decisões de arquitetura devem ser feitas com uma compreensão de todo o sistema. Isso implica uma descrição da maioria dos requisitos do sistema. Para verificar a arquitetura, você implementa um sistema que demonstre as escolhas de arquitetura e execute casos de uso significativos.

No final da fase de elaboração, você examina o escopo e os objetivos detalhados do sistema, a escolha de arquitetura e a solução para os principais riscos, além de decidir se deve prosseguir com a construção.

3.2.3.1.3 Construção

Durante a fase de construção, você desenvolve, de maneira iterativa e incremental, um produto completo, pronto para a transição à sua comunidade de usuários. Isso implica uma descrição dos requisitos restantes e de critérios de aceitação, dando corpo ao projeto e concluindo a implementação e o teste do software.

No final da fase de construção, você decide se o software, ambiente e usuários estão todos prontos para se tornarem operacionais.

3.2.3.1.4 Transição

Durante a fase de transição, você torna o software disponível à comunidade de usuários. Após o sistema ser colocado nas mãos de seus usuários finais, sempre surgem questões que requerem algum desenvolvimento adicional, com a finalidade de ajustar o sistema, corrigir alguns problemas identificados ou concluir algumas características propostas *a posteriori.* Essa fase tipicamente é iniciada com uma versão beta do sistema, que depois é substituída pelo sistema de produção.

No final da fase de transição, você decide se foram alcançados os objetivos do ciclo de vida do projeto e determina se deverá iniciar outro ciclo de desenvolvimento. Esse também é um ponto em que as lições aprendidas no projeto deverão ser assimiladas para aprimorar o processo de desenvolvimento e serem aplicadas no próximo projeto.

3.2.3.2 Iterações

Cada fase do RUP ainda pode ser dividida em iterações. Uma iteração é um ciclo completo de desenvolvimento, resultando em uma versão (interna ou externa)

de um produto executável que constitui um subconjunto do produto final em desenvolvimento e cresce de modo incremental de uma iteração para outra para se tornar o sistema final. Cada iteração passa pelos vários fluxos de trabalho do processo, embora com uma ênfase diferente em cada um deles, dependendo da fase. Durante a concepção, o foco está na captação de requisitos. Durante a elaboração, o foco passa a ser a análise e o projeto. A implementação é a atividade central na construção e a transição, na entrega.

3.2.4 Ciclos de desenvolvimento

A passagem pelas quatro principais fases é chamada um ciclo de desenvolvimento e resulta na geração de um software. O primeiro passo das quatro fases é chamado o ciclo inicial de desenvolvimento. A menos que a vida do produto seja interrompida, um produto existente evoluirá para sua próxima geração, pela repetição da mesma seqüência de fases de concepção, elaboração, construção e transição. Isso é a evolução do sistema, de modo que os ciclos de desenvolvimento posteriores aos ciclos iniciais são seus ciclos de evolução.

O *Rational Unified Process* é composto por nove fluxos de trabalho de processo:

Quadro 3: Fluxos de Trabalho do Processo

1.Modelagem de negócio	Descreve a estrutura e a dinâmica da empresa.
2.Requisitos	Descreve o método baseado em casos de uso para identificar requisitos
3.Análise e projeto	Descreve as várias visões da arquitetura.
4.Implementação	Leva em consideração o desenvolvimento do software, o teste da unidade e a integração.
5.Teste	Descreve os casos de teste, procedimentos e medidas para acompanhamento de erros.
6.Utilização	Abrange a configuração do sistema a ser entregue.
7.Gerenciamento de mudança e configuração	Controla as modificações e mantém a integridade dos artefatos do projeto.
8.Gerenciamento de projeto	Descreve várias estratégias para o trabalho com um processo iterativo.

9.Ambiente	Abrange a infra-estrutura para o desenvolvimento do sistema.

Fonte: Kruchten (2000)

Capturado em cada fluxo de trabalho de processo está um conjunto de atividades e artefatos correlacionados. Um artefato é algum documento, relatório ou executável, que é produzido, manipulado ou consumido. Uma atividade descreve as tarefas — criando, realizando e verificando etapas — executadas pelos trabalhadores para criar ou modificar artefatos, juntamente com as técnicas e diretrizes para a realização de tarefas, possivelmente incluindo a utilização de ferramentas para ajudar a automação de algumas das tarefas.

Conexões importantes entre os artefatos estão associadas a alguns desses fluxos de trabalho de processo. Por exemplo, o modelo de caso de uso gerado durante a captação de requisitos é realizado pelo modelo de projeto a partir do processo de análise e estruturação, implementado pelo modelo de implementação a partir do processo de implementação e verificado pelo modelo de teste a partir do processo de teste.

3.2.5 Artefatos

Cada atividade do RUP tem artefatos associados, ou exigidos como uma entrada ou gerados como uma saída. Alguns artefatos são utilizados com a finalidade de direcionar a entrada para atividades subseqüentes, mantidos como recursos de referência sobre o projeto ou gerados em um formato como entregas contratuais.

3.2.5.1 Modelos

Os modelos são o tipo mais importante de artefato do RUP. Um modelo é uma simplificação da realidade, criado para proporcionar uma melhor compreensão do sistema que está sendo criado. Existem nove modelos que, em conjunto, abrangem todas as decisões importantes para a visualização, especificação, construção e documentação de um sistema complexo de software.

Quadro 4: Tipos de Modelos do RUP

1.Modelo de Negócio	Estabelece uma abstração da empresa.
2.Modelo de Domínio	Estabelece o contexto do sistema.
3.Modelo de Caso de Uso	Estabelece os requisitos funcionais do sistema.
4.Modelo de Análise (opcional)	Estabelece o projeto de uma idéia.
5.Modelo de Projeto	Estabelece o vocabulário do problema e de sua solução.
6.Modelo de Processo (opcional)	Estabelece os mecanismos de concorrência e de sincronização do sistema.
7.Modelo de Implantação	Estabelece a topologia do hardware em que o sistema é executado.
8.Modelo de Implementação	Estabelece as partes utilizadas para montar e liberar o sistema físico.
9.Modelo de Teste	Estabelece os caminhos pelos quais o sistema é validado e verificado.

Fonte: Kruchten (2000)

3.2.5.2 Visões

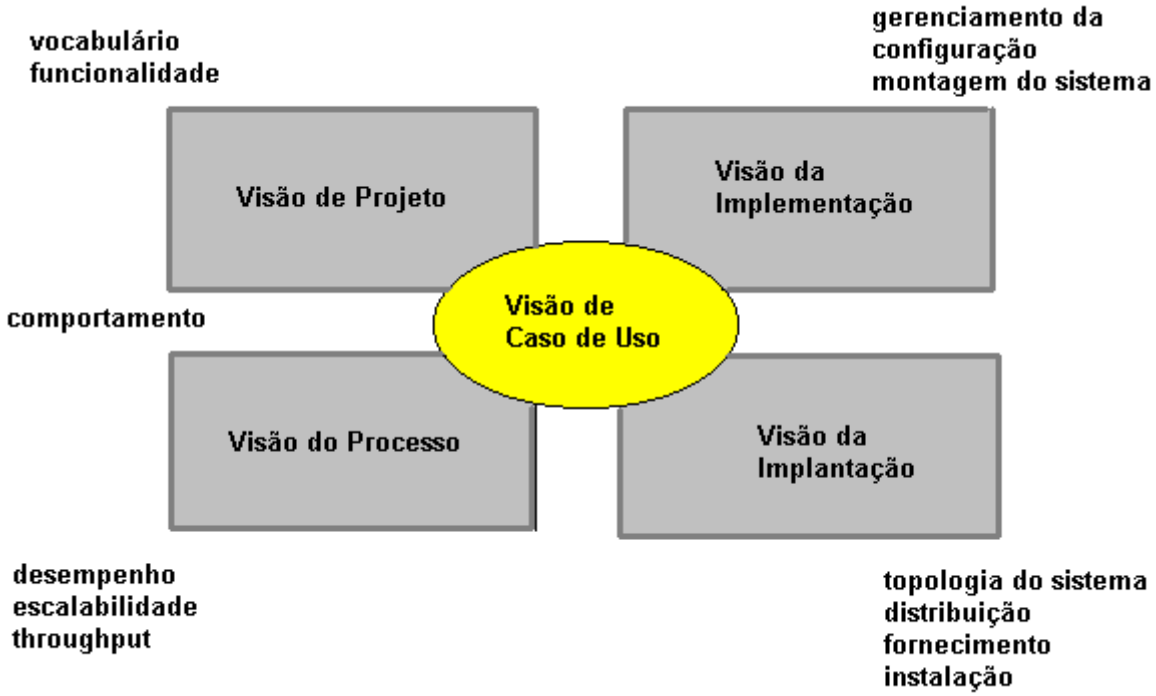
Uma visão é uma projeção em um modelo. No *Rational Unified Process*, a arquitetura de um sistema é capturada em cinco visões interligadas:

Quadro 5: As 4 + 1 visões da modelagem da arquitetura de um sistema

1. Visão do Projeto	Abrange as classes, interfaces e colaborações que formam o vocabulário.
2.Visão do Processo	Abrange o desempenho, a escalabilidade e o *throughput.*
3.Visão de Implantação	Abrange a topologia de hardware do sistema
4.Visão de Implementação	Abrange o gerenciamento de configuração das versões de componentes e arquivos
5.Visão de Casos de Uso	Especifica o comportamento do sistema visto pelos seus usuários.

Fonte: Kruchten (2000)

Figura 3: As 4 + 1 Visões da Modelagem da Arquitetura de um Sistema



Fonte: Kruchten (2000)

3.2.5.2.1 Visão de Caso de Uso

A visão de caso de uso abrange os casos do uso que descrevem o comportamento do sistema conforme é visto pelos seus usuários finais, analistas e pessoal de teste. Essa visão não especifica realmente a organização do sistema de um software. Porém, ela existe para especificar as forças que determinam a forma da arquitetura do sistema. Com a UML, os aspectos estáticos dessa visão são capturados em diagramas de caso de uso, enquanto os aspectos dinâmicos são capturados em diagramas de interação, diagramas de gráfico de estados e diagramas de atividades.

3.2.5.2.2 Visão de Projeto

A visão de projeto de um sistema abrange as classes, interfaces e colaborações que formam o vocabulário do problema e de sua solução. Essa perspectiva proporciona principalmente um suporte para os requisitos funcionais do sistema, ou seja, os serviços que o sistema deverá fornecer a seus usuários finais. Com a UML, os aspectos estáticos dessa visão são captados em diagramas de classes e de objetos; os aspectos dinâmicos são captados em diagramas de

interações, diagramas de gráfico de estados e diagramas de atividades.

3.2.5.2.3 Visão do Processo

A visão do processo abrange os *threads* e os processos que formam os mecanismos de concorrência e de sincronização do sistema. Essa visão cuida principalmente de questões referentes ao desempenho, à escalabilidade e ao *throughput* do sistema. Com a UML, os aspectos estáticos e dinâmicos dessa visão são captados nos mesmos tipos de diagramas da visão de projeto, mas com o foco voltado para as classes ativas que representam esses *threads* e processos.

3.2.5.2.4 Visão de Implementação

A visão de implementação de um sistema abrange os componentes e os arquivos utilizados para a montagem e fornecimento do sistema físico. Essa visão envolve principalmente o gerenciamento da configuração das versões do sistema, compostas por componentes e arquivos de alguma maneira independentes, que podem ser reunidos de diferentes formas para a produção de um sistema executável. Com a UML, os aspectos estáticos dessa visão são capturados em diagramas de componentes; os aspectos dinâmicos são capturados em diagramas de interações, de gráfico de estados e de atividades.

3.2.5.2.5 Visão de Implantação

A visão de implantação de um sistema abrange os nós que formam a topologia de hardware em que o sistema é executado. Essa visão direciona principalmente a distribuição, o fornecimento e a instalação das partes que constituem o sistema físico. Com a UML, os aspectos estáticos dessa visão são capturados em diagramas de implantação; os aspectos dinâmicos são capturados em diagramas de interações, de gráfico de estados e diagramas de atividades.

Cada uma dessas cinco visões pode ser considerada isoladamente, permitindo que diferentes participantes dirijam seu foco para os aspectos da arquitetura do sistema que mais lhes interessam. Essas cinco visões também interagem entre si - os nós na visão de implantação contém componentes da visão de implementação que, por sua vez, representa a realização física de classes, interfaces, colaborações e classes ativas provenientes das visões de projeto e de processo. A UML permite expressar cada uma dessas cinco visões e suas

interações.

A UML é amplamente independente do processo. Isso significa que não se limita ao ciclo de vida de desenvolvimento de determinado software. Porém, para obter o máximo proveito da UML, será preciso levar em consideração um processo com as seguintes características: orientado a caso de uso, centrado na arquitetura, interativo e incremental.

3.2.5.3 Outros artefatos

Os artefatos do *Rational Unified Process* são categorizados como artefatos de gerenciamento ou artefatos técnicos. Os artefatos técnicos podem ser divididos em quatro conjuntos principais:

Quadro 6: Artefatos Técnicos do RUP

1.Conjunto de Requisitos	Descreve o que o sistema deve fazer.
2.Conjunto de Projeto	Descreve como o sistema é construído.
3.Conjunto de Implementação	Descreve a montagem dos componentes do software desenvolvido.
4.Conjunto de Implantação	Fornece todos os dados para a configuração de implantação.

Fonte: Kruchten (2000)

3.2.5.3.1 Conjunto de Requisitos

Agrupa todas as informações descrevendo o que o sistema deve fazer. Poderá incluir um modelo de caso de uso, um modelo de requisitos não-funcionais, um modelo de domínio, um modelo de análise e outras formas de expressão das necessidades do usuário, incluindo, mas sem estar limitado a simulações, protótipos de interface, restrições regulatórias e assim por diante.

3.2.5.3.2 Conjunto de Projeto

Agrupa informações descrevendo como o sistema é construído e captura as decisões referentes ao modo como o sistema é construído, levando em consideração todas as restrições de tempo, orçamento, herança, reutilização, objetivos de qualidade e assim por diante. Isso pode abranger um modelo de projeto, um modelo de teste e outras formas de expressão da natureza do sistema, incluindo, mas sem estar limitado a protótipos e arquiteturas executáveis.

3.2.5.3.3 Conjunto de Implementação

Agrupa todas as informações sobre os elementos de software que compõem o sistema, incluindo, mas sem estar limitado a código-fonte em várias linguagens de programação, arquivos de configuração, arquivos de dados, componentes de software e assim por diante, juntamente com as informações que descrevem como montar o sistema.

3.2.5.3.4 Conjunto de Implantação

Agrupa todas as informações sobre a forma como o software é atualmente empacotado, entregue, instalado e executado no ambiente de destino.

3.3 Metodologia VINCIT

3.3.1 Introdução

A Metodologia VINCIT é um produto da empresa FUZION Engenharia de Software Ltda em parceria com a Sterling Software e a Computer Associates. Trata-se de uma Metodologia de Engenharia de Software que propõe um processo de construção de software orientado a objetos, que descreve inúmeras técnicas para inúmeros perfis de profissionais, possibilitando o envolvimento de diversos especialistas na produção de um software baseado em componentes.

Como um processo, é composto de métodos, que por sua vez, trabalham com linguagens para chegar a um determinado resultado. Um conjunto de símbolos com uma semântica definida e uma sintaxe determinada constitui o entendimento da Vincit em relação a uma linguagem. Contudo a elaboração dos modelos, ou o conjunto de símbolos interligados, que representam o que deverá ser construído através da linguagem, não faz parte da mesma. Ou seja, o aprendizado de como se construir algo significativo, está fora do escopo da definição da linguagem. Nesse caso, a VINCIT utiliza a linguagem de modelagem UML, que não contém instruções de como construir os modelos, mas somente a notação a ser utilizada para esse fim.

A VINCIT possui um conjunto de métodos, que por sua vez caracteriza um processo, organizados de forma consistente, distribuídos no tempo e com um perfeito acoplamento de entrada e saída. Os métodos representam as tarefas

executadas, sendo que um insumo em um método pode ser o produto gerado em outro método ou vir direto da entrada, com organização e encadeados no tempo.

A VINCIT é um processo de produção de software iterativo, incremental e controlado, onde a entrada representa o negócio a ser informatizado e a saída é o software produzido.

O processo VINCIT possui quatro camadas: Ciclo, Fase, Atividade e Tarefa. Um ciclo possui várias fases, uma fase possui várias atividades e uma atividade possui várias tarefas.

Um ciclo representa a separação semântica natural dos passos que compõe o desenvolvimento de um software e são os seguintes:

I Ciclo de Requisitos

II Ciclo de Análise

III Ciclo de Projeto

IV Ciclo de Implementação

Uma fase representa em que parte de um ciclo se encontra o desenvolvimento do software. As fases são:

1. Concepção
2. Elaboração
3. Construção
4. Liberação

Uma atividade representa os tipos de métodos que estão sendo aplicados dentro de uma fase. As atividades são:

A Especificação

B Particionamento

C Componentização

D Programação

Uma tarefa representa os métodos utilizados para transformar insumos em produtos e são diferentes, dependendo da atividade que ela pertence. Desta maneira, para identificar em que ponto do desenvolvimento se encontra um sub-sistema, precisamos de quatro coordenadas: **ciclo, fase, atividades e tarefa**.

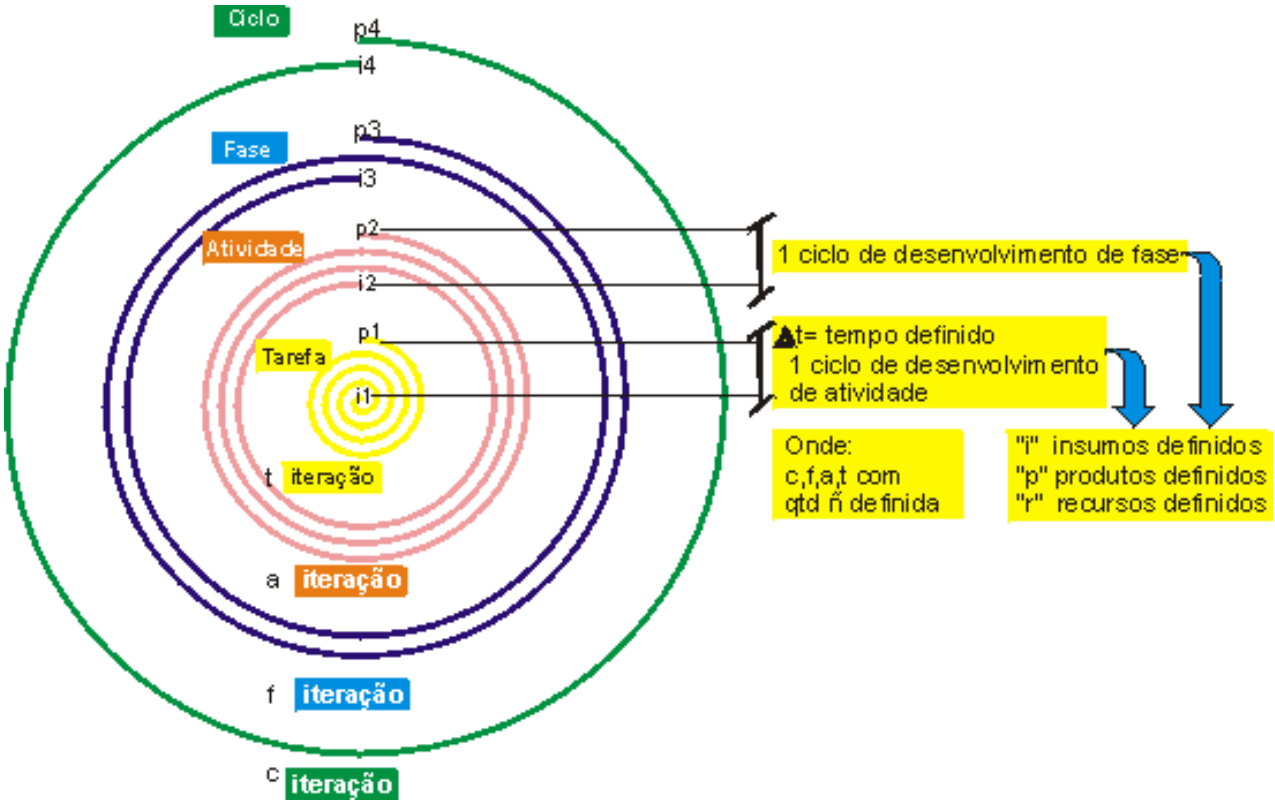
Embora, inicialmente, a divisão da VINCIT em quatro camadas possa parecer inviável, seu uso prático é extremamente facilitado por esta divisão.

Agora precisamos integrar os atributos do processo de produção de software: **Interativo, Incremental e Controlado.**

A VINCIT não está orientada a iterações e sim a produtos. Isto não significa que não se utilize das iterações para produzir um software, significa que o foco está no produto gerado por um conjunto de iterações e não na gerência particular das iterações.

Um ciclo de desenvolvimento, mostrado na figura 3.3, é definido como sendo um conjunto de “n” iterações necessárias para produzir um conjunto “p” de produtos, através de um conjunto “i” de insumos dentro de uma camada específica da VINCIT. Sendo assim, um ciclo de desenvolvimento possui “n” iterações não definidas, em um tempo “t” definido, utilizando um insumo “i” definido e gerando um produto “p” definido, através de um conjunto de recursos “r” definido. Aliando-se um processo indutivo e incremental a técnicas reais de produção de produtos a partir de insumos, em um ambiente real de produção de software, chegamos à estrutura em camadas da VINCIT e a produção de um software de forma iterativa, incremental e controlada.

Figura 4: O ciclo de desenvolvimento Vincit.



Fonte: CD-ROM da Metodologia Vincit

3.3.2 Composição da metodologia VINCIT

A metodologia Vincit é composta por quatro ciclos, a saber:

Quadro 7: Ciclos da metodologia Vincit

CICLO	OBJETIVO
Requisitos	Neste ciclo os profissionais envolvidos deverão obter o entendimento real do problema e das metas do cliente, e levantar as informações necessárias para o trabalho a ser realizado nos próximos ciclos. O objetivo geral é: • Entender e documentar os Processos de Negócio (casos de uso de negócio) que estão envolvidos diretamente com o software a ser desenvolvido, estabelecendo as fronteiras do sistema com os mesmos; • Entender e documentar os Requisitos Funcionais e Não Funcionais do sistema; • Descrever os casos de Uso relacionados com o sistema.
Análise	No ciclo de análise é feita uma descrição arquitetônica do sistema na perspectiva de um usuário. É enfatizada a descrição do domínio e do comportamento externo visível. No modelo de objetos são descritos os conceitos do domínio do problema e as relações entre eles. O dicionário de dados é produzido de forma incremental procurando relações, atributos, *constraints* e cardinalidades. A interface do sistema é definida como um conjunto de operações que respondem aos eventos solicitados pelos usuários. Os modelos de análise são inspecionados completamente para se verificar inconsistências entre eles e se todos os requisitos do sistema estão atendidos.
Projeto	Este ciclo tem como objetivo iniciar o projeto dos subsistemas definidos no ciclo de análise. Estes subsistemas são pacotes que compõem apenas uma parte do sistema que está sendo desenvolvido. Cada equipe de projetista deverá atuar na especificação e na completa definição do pacote e dos elementos de modelagem que o constituem. As últimas questões quanto ao domínio do problema deverão logo ser atacadas e o projetista

	deverá iniciar o processo de implementação real do sistema.
Implementação	Este ciclo tem como objetivo efetuar a construção de todos os componentes descobertos e modelados nos ciclos anteriores, visando atender as especificações do projeto, que tem como base os requisitos funcionais ou não do negócio do cliente. Cada subsistema deverá gerar os componentes e suas interações construídos, testados e homologados.

Fonte: CD-ROM da Metodologia Vincit

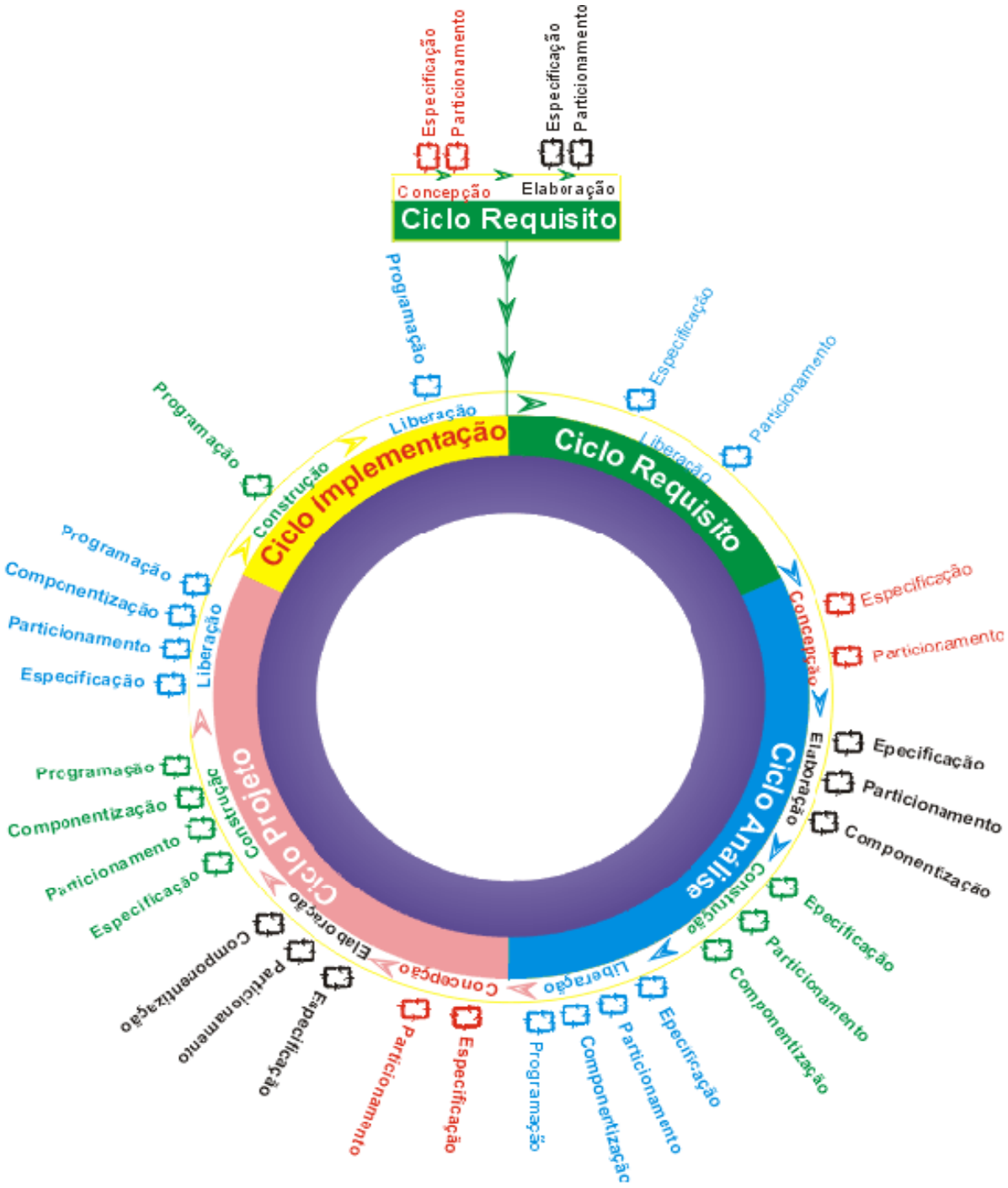
Cada ciclo da metodologia VINCIT é subdividido em até 4 fases, que representam a situação em que se encontra o modelo:

Quadro 3.7: Fases da metodologia Vincit

FASE	OBJETIVO
Concepção	Nesta fase deve-se procurar entender e mapear o problema a ser resolvido, a nível conceitual, lógico e físico.
Elaboração	Esta fase é responsável pela confecção dos modelos, a nível conceitual e lógico, inserindo o software a ser produzido no contexto real das arquiteturas existentes na empresa.
Construção	È a fase responsável pela construção de parte do sistema no ambiente de desenvolvimento.
Liberação	È a fase responsável pela liberação de produtos para o próximo ciclo e também pelo teste e validação dos produtos gerados.

Fonte: CD-ROM da Metodologia Vincit

Figura 5: Os ciclos da metodologia Vincit



Fonte: CD-ROM da Metodologia Vincit

Cada fase da metodologia VINCIT é subdividida em até quatro atividades, que representam a situação em que se encontra o modelo do sistema:

Quadro 9: Atividades da Metodologia Vincit

ATIVIDADE	OBJETIVO
Especificação	È nesta atividade que normalmente se define o que precisa ser feito e se faz um planejamento da fase.
Particionamento	É nesta atividade que normalmente se divide o sistema em sub-sistemas.
Componentização	É nesta atividade que normalmente se define o que pode ou não pode ser componentizado e se projeta os componentes.
Programação	É nesta atividade que normalmente se constrói fisicamente os produtos que compõem o software.

Fonte: CD-ROM da Metodologia Vincit

Cada atividade da metodologia VINCIT poderá possuir tarefas, que correspondem a ações a serem realizadas por um determinado papel de profissional. Cada tarefa terá uma descrição de seu objetivo, acompanhada da indicação das técnicas e insumos necessários para a sua execução, dos produtos por ela gerados, das ferramentas que suportam a sua execução e dos recursos que participam da mesma. Como tarefas poderão existir, entre outras:

- Coordenação do projeto
- Análise do negócio
- Análise de requisito
- Arquitetura de software
- Análise do sistema
- Arquitetura do sistema
- Catalogação de soluções
- Modelagem de componentes
- Implementação do banco de dados
- Implementação da interface do usuário
- Homologação

A metodologia VINCIT proporciona aos seus usuários um conjunto de formulários preparados para a utilização em todas as fases de desenvolvimento do sistema. Estes formulários e procedimentos de utilização constam da documentação da metodologia e poderão ser facilmente adaptados para a empresa cliente.

Todo o roteiro de utilização da metodologia consta em um CDROM fornecido pela empresa proprietária do VINCIT ao se adquirir a metodologia.

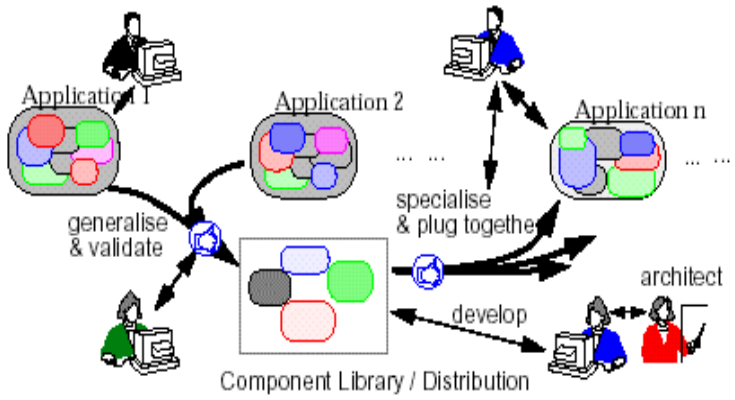
3.4. Metodologia CATALYSIS

3.4.1 Introdução

A metodologia Catalysis faz parte de uma nova geração de metodologias de desenvolvimento de sistemas baseadas em componentes e focadas no negócio, que utiliza a notação padrão da indústria *Unified Modeling Language* (UML). Em desenvolvimento e aplicação desde 1992, foi usada por grandes companhias nos campos de finanças, telecomunicações, seguros, indústrias, controle de processos, simulação de vôo, viagens, transportes e administração de sistemas. Catalysis é uma metodologia não-proprietária, em uso em muitos projetos e apoiada por ferramentas, produtos e serviços de várias companhias. Catalysis é uma marca de serviços da empresa ICON Computing, uma subsidiária da empresa Platinum Technology.

Na metodologia Catalysis, especificada no livro ***Objects, Components, and Frameworks with UML: The Catalysis Approach*** de Desmond Francis D'Souza e Alan Cameron Wills (D'Souza, 1999), o desenvolvimento baseado em componentes enfoca o desenvolvimento de aplicações, no qual os artefatos de código executável, as especificações, arquiteturas e modelos empresariais, são conectados em diferentes escalas, para formar desde aplicações completas até componentes individuais. Os componentes podem ser construídos ajuntando-se, adaptando-se e interligando-se componentes existentes em uma variedade de configurações diferentes. A Catalisys provê apoio integral para a modelagem de componentes do negócio, o projeto da interface, a arquitetura de componentes, o reuso do projeto e do modelo de componentes, além de código.

Figura 6: Visão Catalysis do desenvolvimento baseado em componentes.



Fonte: D'Souza (1999)

3.4.2. As metas da metodologia Catalysis

O enfoque do desenvolvimento baseado em componentes tem um impacto significativo em como os sistemas são projetados, desenvolvidos e implementados. O impacto se reflete em todo o ciclo de vida dos sistemas aplicativos. A metodologia Catalysis provê suporte para especificação, análise, projeto e implementação em escalas de sistemas empresariais distribuídos. As características do método incluem as seguintes:

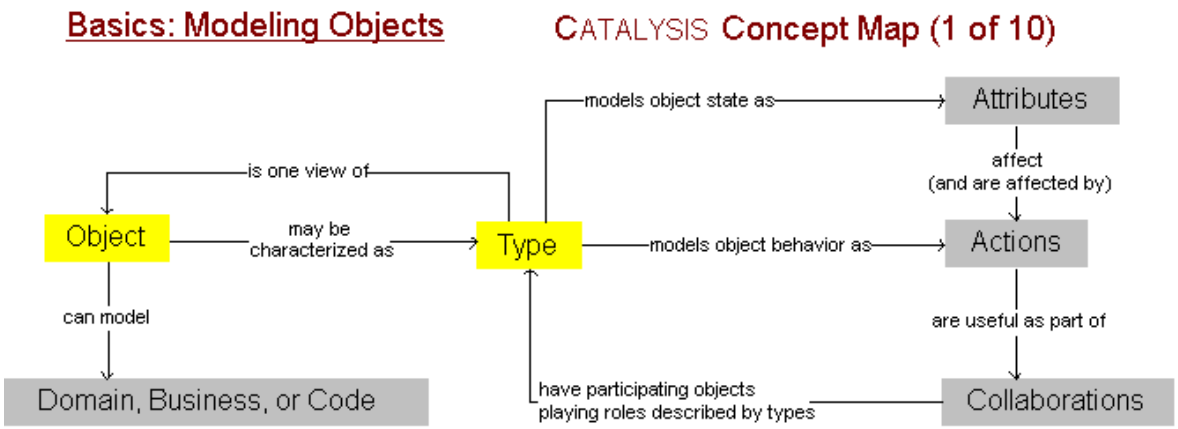
- Os sistemas podem ser modelados como uma coleção de componentes interagindo entre si;
- O comportamento do sistema pode ser analisado em função das interfaces dos componentes;
- Especificações de componentes podem ser descritas independentemente da implementação dos referidos componentes;
- Uma notação precisa e formal deve ser utilizada para descrever as especificações dos componentes, como a UML;
- Padrões de componentes de interações podem ser modelados e conseqüentemente reusados nos novos projetos;
- Um rigoroso processo de refinamento deve ser seguido para relatar o comportamento do sistema em todos os detalhes possíveis.

3.4.3 Mapa de Conceitos Catalysis

Alguns conceitos da metodologia Catalysis devem ser inicialmente definidos para um melhor entendimento do método e permitir uma melhor captura do comportamento do sistema para o nível de abstração desejado.

A metodologia fornece um mapa de conceitos importantes, relacionados no fluxo abaixo, que deve ser seguido de forma a um melhor entendimento.

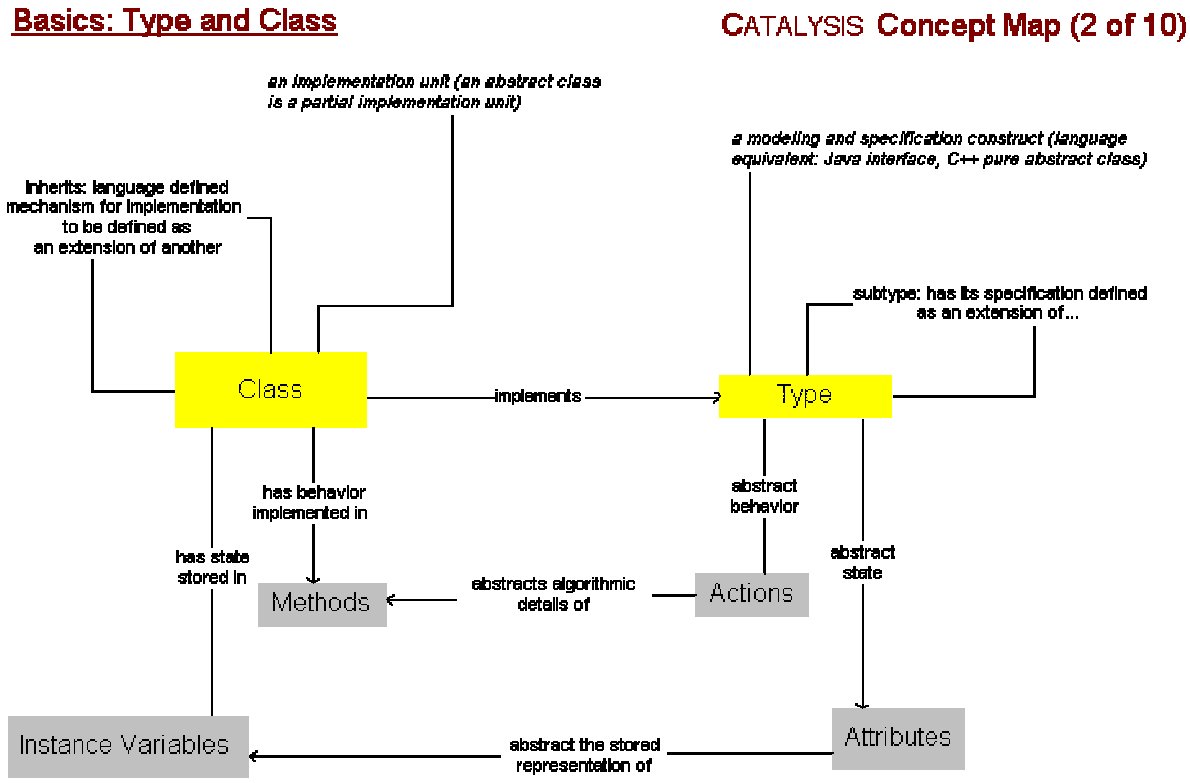
Figura 7: Catalysis Mapa de Conceitos 1 de 10: Modelagem de objetos



Fonte: D'Souza (1999)

Objetos podem ser usados para modelar, conceitualizar e compreender muitos níveis de operações de negócios, domínios de problemas gerais, requerimentos de softwares, entre outros. A descrição de alguns aspectos interessantes do objeto, numa visão parcial e abstrata do objeto é chamada de **Tipo**. Um tipo descreve o comportamento de um objeto usando o estado abstrato dos seus **atributos,** as **ações** que o objeto participa e os efeitos nos atributos. Uma ação é muito útil e importante no contexto de outras ações relacionadas; em combinação elas definem uma **colaboração** que realiza alguma meta.

Figura 8. Catalysis Mapa de Conceitos 2 de 10: Tipos e classes

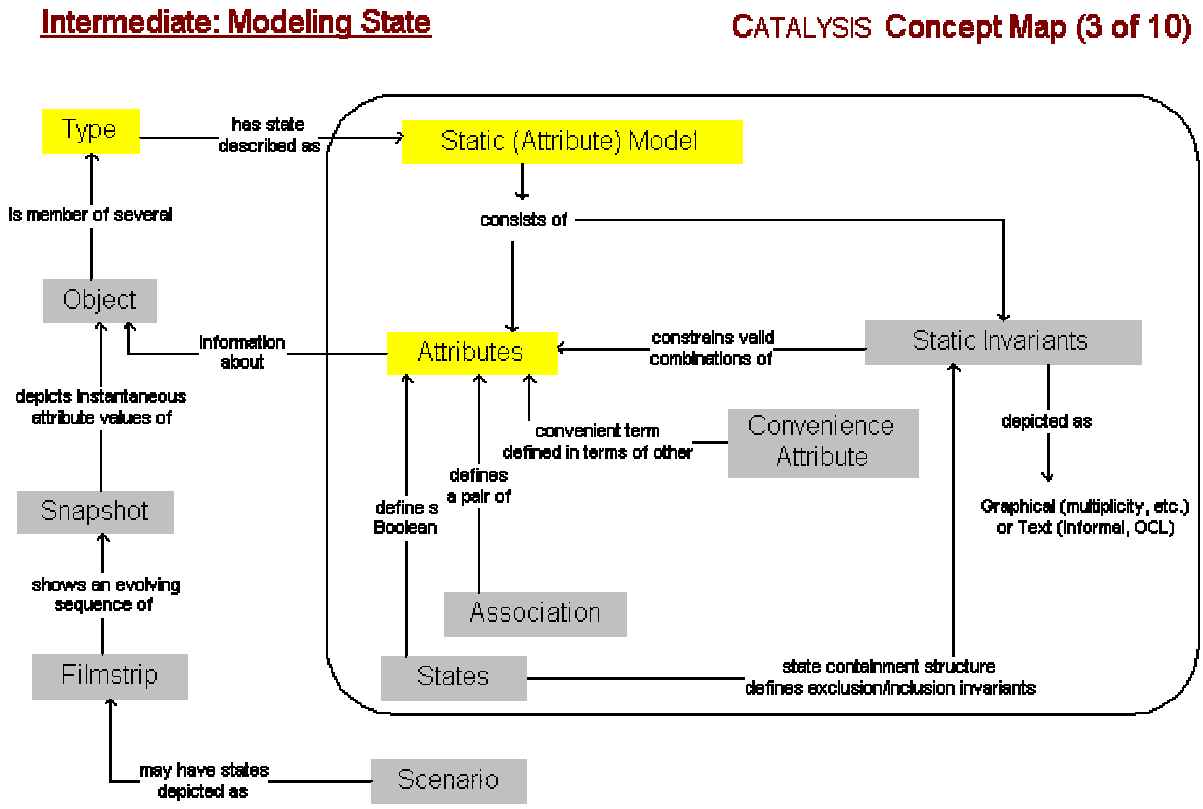


Fonte: D'Souza (1999)

Uma **classe** é uma unidade de implementação de programação para todas as instâncias da classe. Define os **métodos,** procedimentos das suas instâncias, e os dados armazenados como **variáveis de instância**. Cada método pode ler e pode modificar as variáveis de instância dessa instância e pode se comunicar com outros objetos, se referidos por essas variáveis de instância, ou pode ser passado como parâmetro para o método.

Um **tipo** é a modelagem e a especificação de uma construção em código de programação OO, mas também pode ser um algoritmo abstrato ou um comportamento procedimental com especificações de **ações**, resumindo dados ou a representação da informação por **atributos.** Cada implementação de um método deve definir o resultado líquido especificado pela ação e os dados devem representar a informação armazenada. Uma classe implementa muitos tipos. Cada tipo pode ser implementado por várias classes.

Figura 9: Catalysis Mapa de Conceitos 3 de 10: Modelando estados.



Fonte: D'Souza (1999)

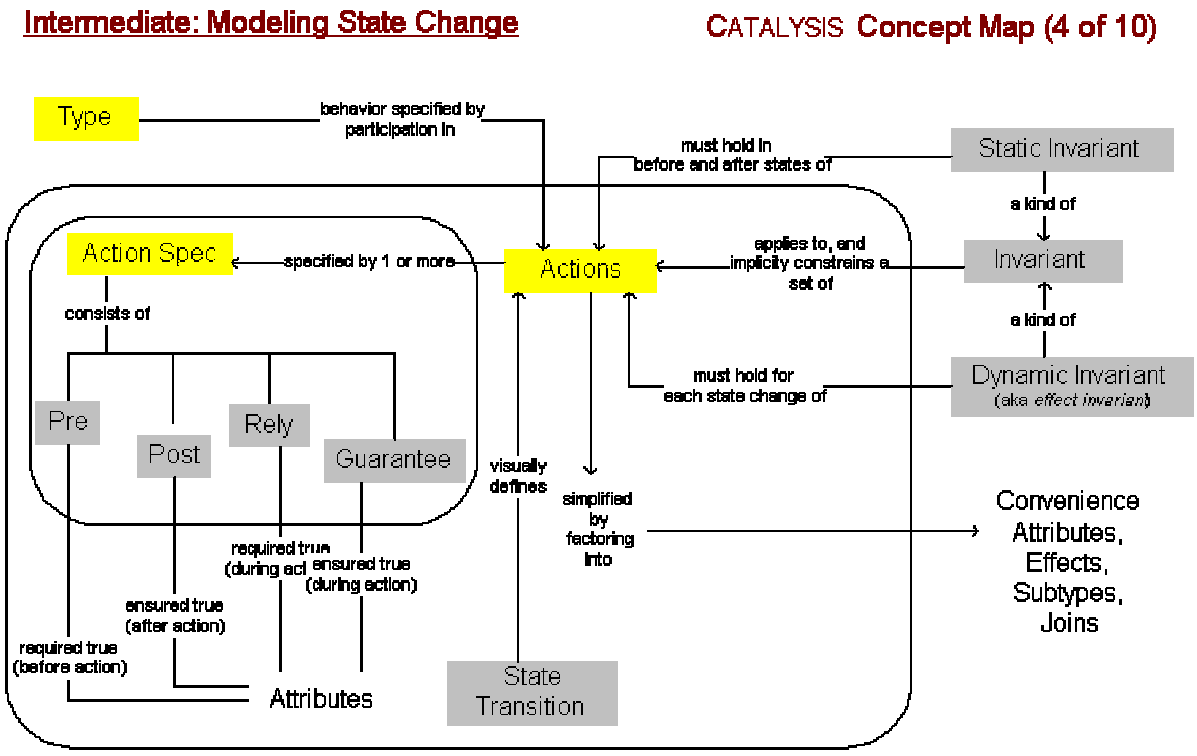
O estado de um **objeto** é definido por um **modelo estático** em um tipo que descreve esse objeto. O modelo estático consiste de um conjunto de **atributos** e um conjunto de **invariantes estáticas** – regras nas quais combinações de valores de atributos devem ser válidas, para esse estado do objeto ser válido. Um **atributo conveniente** é introduzido por conveniência e pode descrever coisas sem verificar os atributos mais básicos, simplificando o que se necessita dizer. Um atributo de conveniência deve ter uma invariante estática que define seu valor nos termos de outros atributos.

Uma **associação** descreve graficamente um par inverso de atributos. A estrutura de **estados** em um diagrama de estado define invariantes estáticas (exclusão, inclusão) entre os correspondentes estados.

Um ***snapshot*** mostra um conjunto de objetos e sua configuração em um determinado tempo (os valores de seus atributos ou ligações a outros objetos). Um

filmstrip mostra uma seqüência de *snapshots* que evoluem com as etapas de um **cenário**.

Figura 10: Catalysis Mapa de Conceitos 4 de 10: Modelando mudanças de estado.



Fonte: D'Souza (1999)

O comportamento de um tipo é especificado por **ações** que participam do próprio tipo. Cada ação é detalhada por uma ou mais **especificações de ações** que definem os efeitos da ação em termos de atributos dos objetos envolvidos e as condições sobre as quais os efeitos atuam, usando-se os seguintes tipos de condições:

- Pré-condições: o que se espera ser verdadeiro no começo da ação.
- Condições de confiança: o que se espera ser verdadeiro e mantido durante a ação.
- Pós-condições: o que é assegurado para se tornar verdadeiro no fim da ação.
- Condições de garantia: o que é assegurado para permanecer verdadeiro durante a ação.

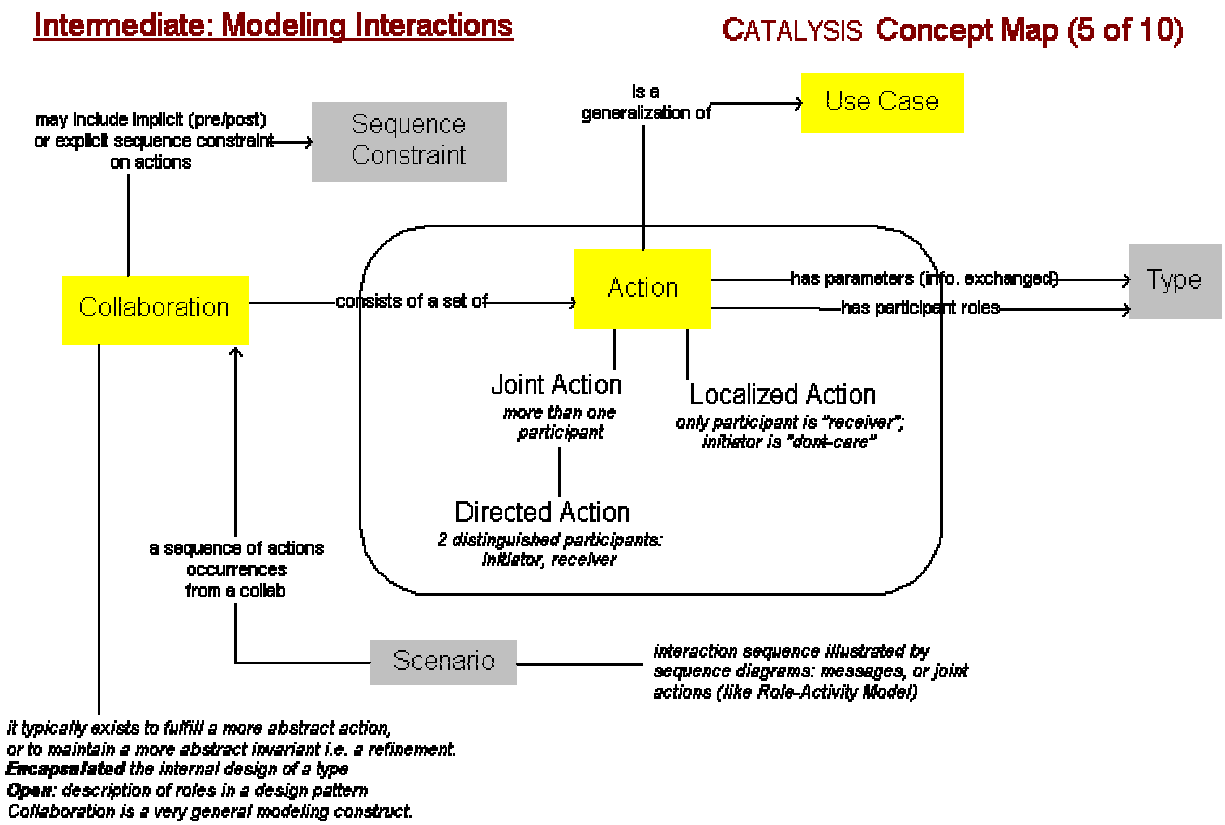
As pós-condições e as condições de garantia devem permanecer durante toda a ocorrência da ação, desde que existam as pré-condições e condições de

confiança. As condições de confiança e de garantia são utilizadas menos freqüentemente e somente para ações não-atômicas com duração significativa, que podem acontecer simultaneamente com outras ações.

As **invariantes** adicionam restrições (*constraints)* implícitas a todas as ações (dentro do "escopo" dessas invariantes). As **invariantes estáticas** devem ser aplicadas nos estados “antes” e “depois” de cada ação e são adicionados implicitamente a todas as pré-condições e pós-condições de todas essas ações. As **invariantes dinâmicas** são aplicadas a cada pós-condição (transição do estado) de todas essas ações.

Uma **transição de estado** é uma representação gráfica de uma especificação de ação. Uma parte significativa dos modelos bem elaborados depende de se usar técnicas simples de especificações de ações, numa forma natural de descrição.

Figura 11: Catalysis Mapa de Conceitos 5 de 10: Modelando interações.



Fonte: D’Souza (1999)

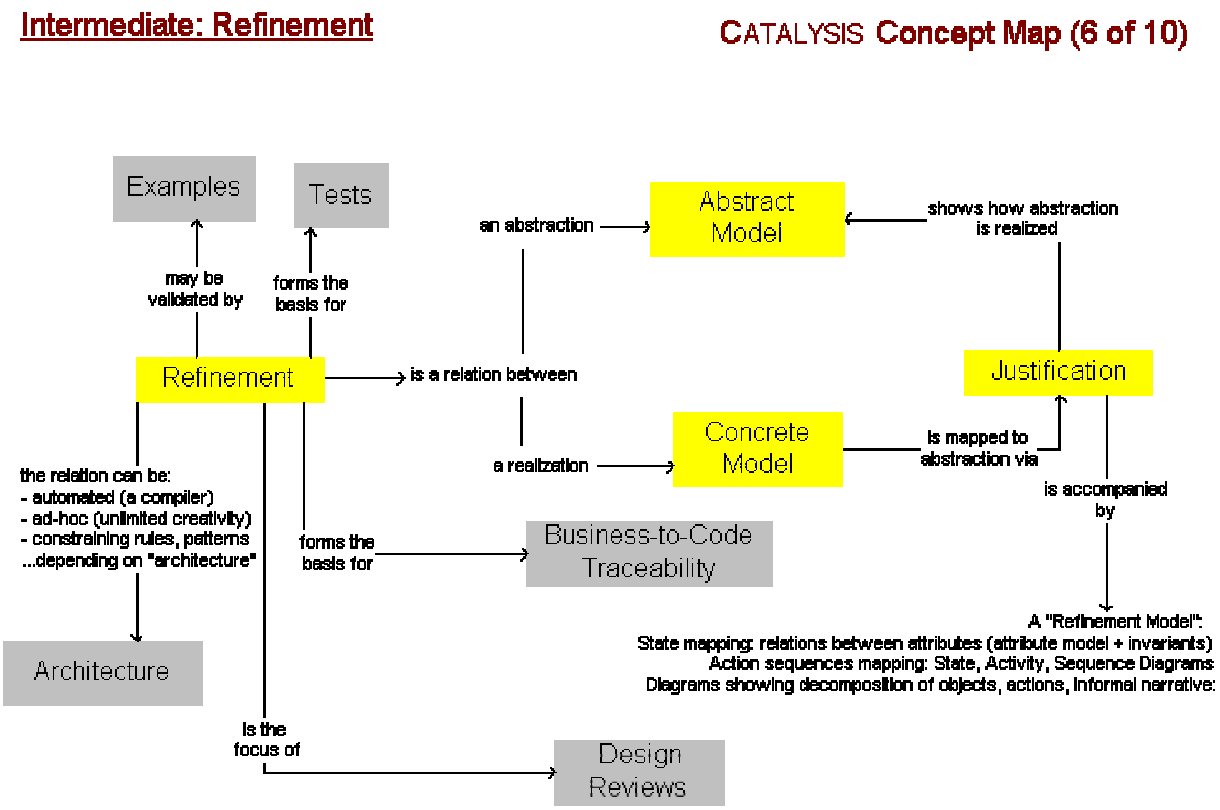
As **ações** são úteis como a parte de **colaborações**. Cada ação tem os objetos participantes e os parâmetros, caracterizados por seus tipos. Uma ação comum tem mais de um participante que a afeta, ou é afetado por essa ação.

As ações, em uma colaboração, podem acontecer somente em determinadas seqüências; assim a colaboração pode ter uma ***sequence_constraint*** (restrição de seqüência). As restrições mais explícitas de seqüência estão no código procedimental, fazendo com que aconteçam exatamente na ordem prescrita.

Um **cenário** é uma ocorrência de uma seqüência de ações em uma colaboração. A maioria das colaborações existe como **refinamentos**, para cumprir uma ação mais abstrata ou para manter uma invariante. O conceito de ação com refinamento generaliza o conceito de **caso do uso**. A construção de casos de uso combina:

- Uma ação abstrata;
- Um conjunto de ações refinadas em uma colaboração;
- Um refinamento que traça uma seqüência de ações mais finas; e
- Construções para ajudar a estruturar essas ações (<estende>, <inclui >, etc..).

Figura 12: Catalysis Mapa de Conceitos 6 de 10: Refinamento.



Fonte: D'Souza (1999)

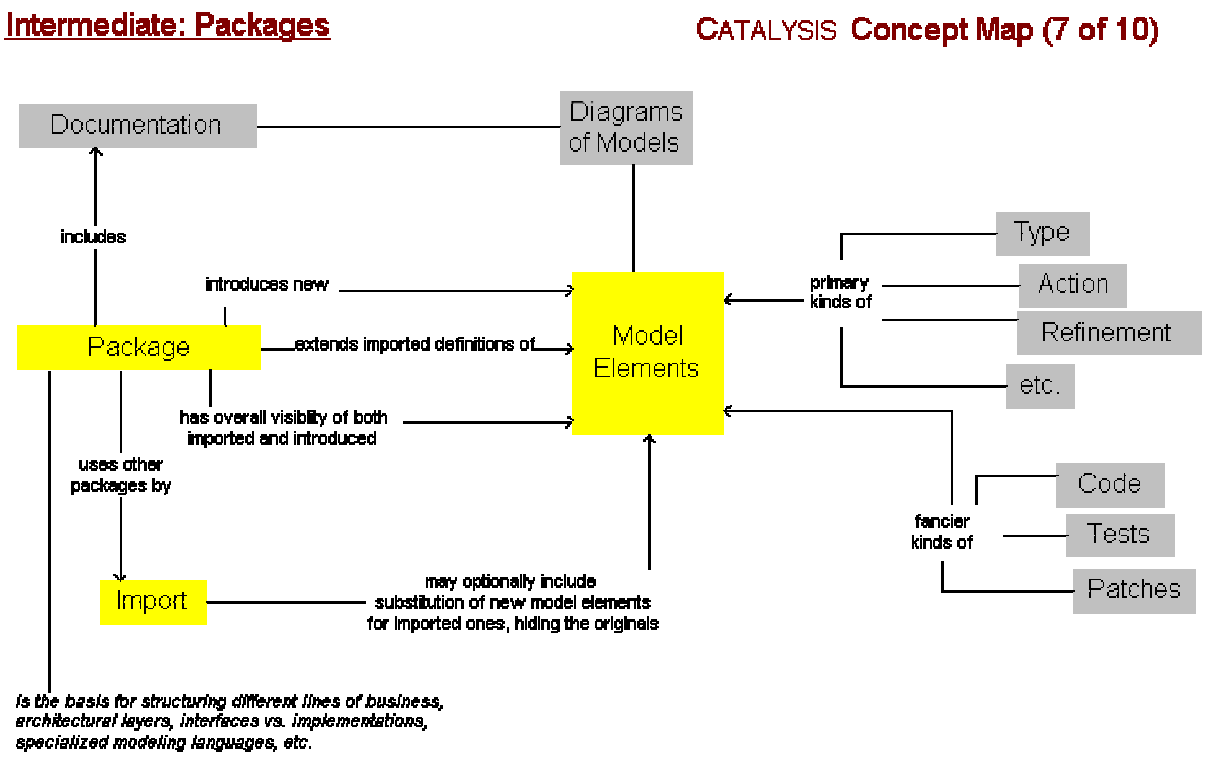
Um refinamento é uma relação entre duas descrições, um **modelo abstrato** e **um modelo concreto**. Tudo que é descrito no modelo abstrato é ainda verdadeiro sobre o modelo concreto, desde que se re-interprete os termos concretos através de algum mapeamento que **justifique** o refinamento. Um refinamento não implica numa seqüência *top-down* do desenvolvimento!

Em particular, um **refinamento**:

- Pode ser validado através de **exemplos**.
- São a base para elaboração de **testes**: cada teste escrito, baseado no modelo concreto (ou de implementação), tem que conhecer a especificação correspondente que foi escrita baseada no modelo abstrato, quando mapeado pelo refinamento.
- É o foco das **revisões do desenho**: a especificação do que se pretende construir deve evoluir para a solução do desenho, através do seu mapeamento.
- São a base para o delineamento (***traceability*)** das regras de negócio para a codificação em uma linguagem de programação.
- Em casos extremos, pode ser usado para verificação formal da correção das rotinas.

Associado com um refinamento corresponde uma **arquitetura** que define o modelo concreto de construção (desenho ou elementos de implementação) e as regras que devem ser aplicadas para construção e que satisfazem qualquer exigência.

Figura 13: Catalysis Mapa de Conceitos 7 de 10: Pacotes.



Fonte: D'Souza (1999)

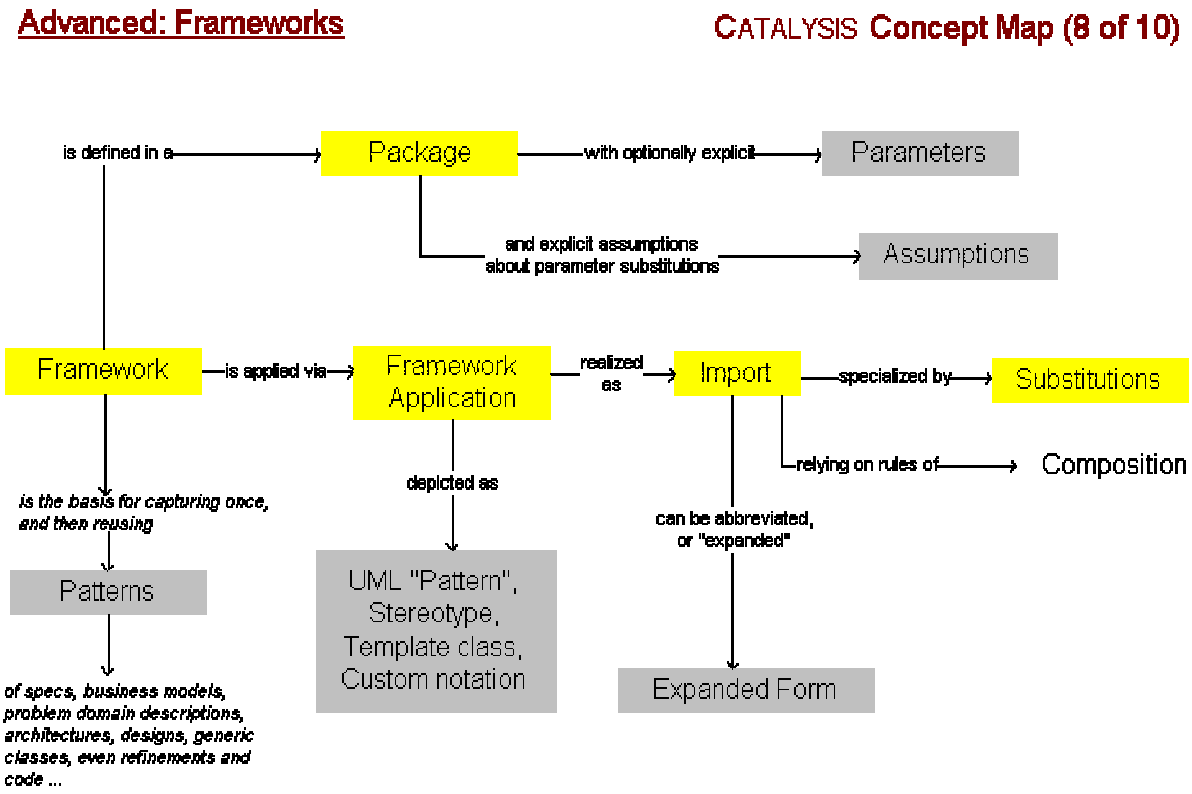
Modelos precisam ser estruturados. Além de estarem divididos em tipos, colaborações, refinamentos, etc. é necessário permitir que grupos de pessoas trabalhem em partes diferentes, com diferentes licenças e visões sobre os objetos comuns, façam administração de configuração e o controle de versão em unidades estruturadas.

Um **pacote** é como um dispositivo de estruturação de modelos. Em um pacote pode-se introduzir novos **elementos de modelo** e também é possível **importar** outro pacote, deixando visível os seus elementos no modelo. Pode-se também definir propriedades adicionais aos modelos dentro de seu pacote. Uma combinação do que foi introduzido localmente, importado de outro lugar e acrescido aos dados importados, define o total do que é visível no pacote.

As descrições narrativas são partes de um pacote, a **documentação** estruturada é também parte de um pacote, incluindo diagramas dos elementos do modelo. Quando são feitas importações, pode-se usar o pacote fonte como um molde (*template),* substituindo alguns de seus elementos. Esta facilidade é usada para definir ***frameworks***.

Os elementos de um modelo incluem os **tipos e ações**, mas podem também incluir o **código**, ***patches* incrementais de códigos, especificações de testes, casos do teste, dados de teste** entre outros.

Figura 14: Catalysis Mapa de conceitos 8 de 10: Frameworks.

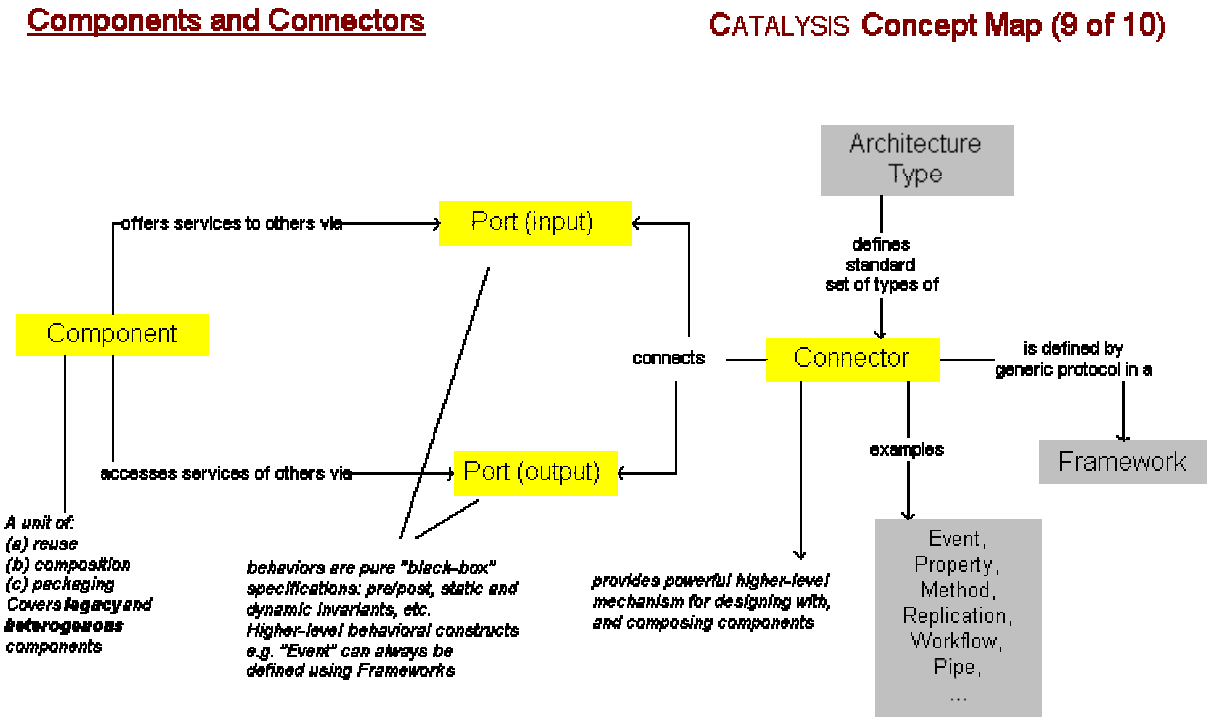


Fonte: D'Souza (1999)

Um ***framework*** é um molde para modelos de negócios, de especificações, de projeto, entre outros, que é definido em um **pacote.** Os **parâmetros** desse molde e todas as **suposições** feitas nesses modelos sobre as **substituições** pretendidas para aqueles parâmetros, podem ser feitas de forma explícita.

Uma **estrutura** é usada através de uma **aplicação de framework**, que corresponde a **importar** um pacote que contem a definição da estrutura, substituindo seus elementos específicos de modelos por parâmetros. Os *frameworks* são úteis para muitos tipos de **padrões** em todos os níveis de modelagem.

Figura 15: Catalysis Mapa de Conceitos 9 de 10: Componentes e conectores.

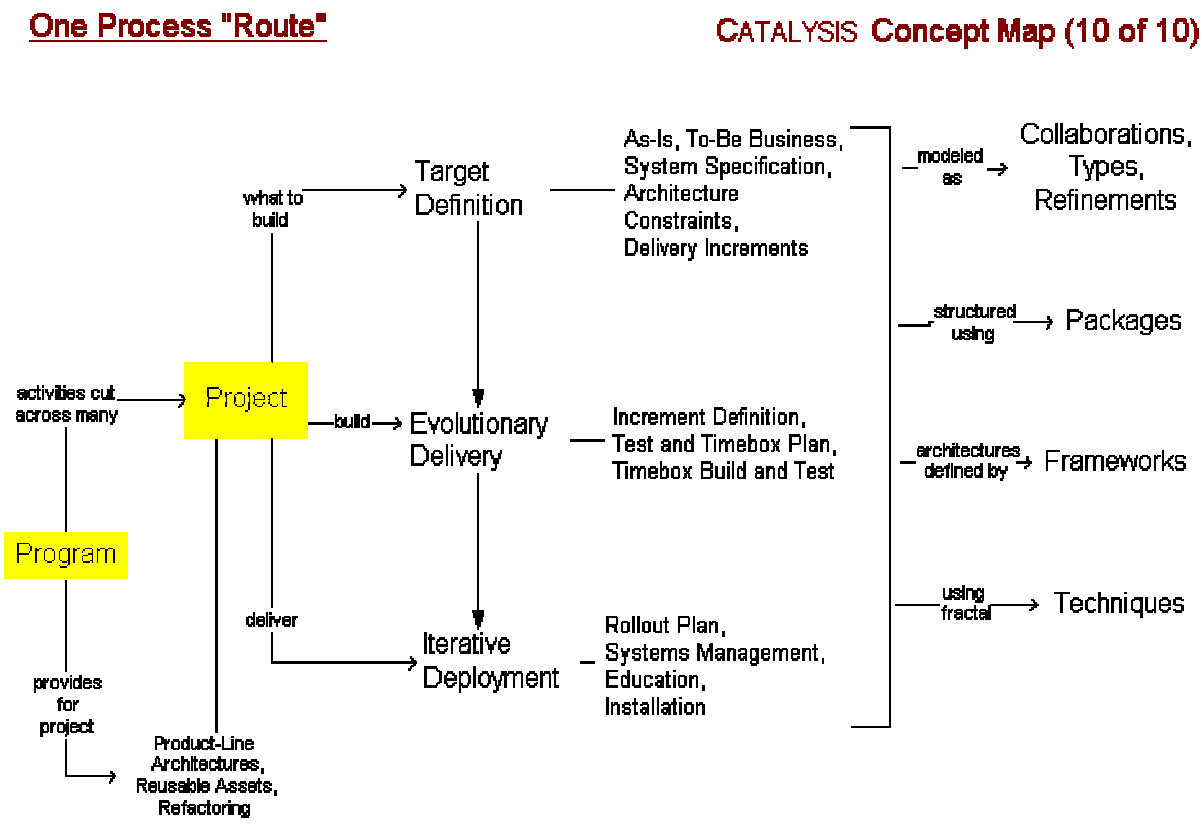


Fonte: D'Souza (1999)

Os **componentes**, com os **conectores** entre suas **portas**, fornecem uma maneira poderosa de abstrair estruturas complexas de objetos e os protocolos de interações entre esses objetos. Muitos tipos de descrições de arquitetura podem ser feitos de forma simples usando-se esses princípios. Estas descrições tendem a ser muito simétricas, isto é, explicitam os serviços fornecidos (abstraído em **portas da entrada**) e serviços requeridos (abstraído em **portas de saída**).

Na metodologia Catalysis, os tipos de conectores são definidos com precisão usando-se ***frameworks***, sendo possível introduzir os seguintes tipos de conectores específicos para o projeto ou para a arquitetura de componentes da organização: evento, propriedade, método, replicação, etc..

Figura 16: Catalysis Mapa de Conceitos 10 de 10: O fluxo do processo.



Fonte: D'Souza (1999)

Existem muitas "rotas" possíveis para um projeto utilizando a metodologia Catalysis, dependendo da natureza de projeto, da experiência da equipe, da cultura incorporada à organização, das dificuldades matrimoniais, fases da lua, etc. Em particular, esboçou-se aqui um projeto passando pelas seguintes estágios:

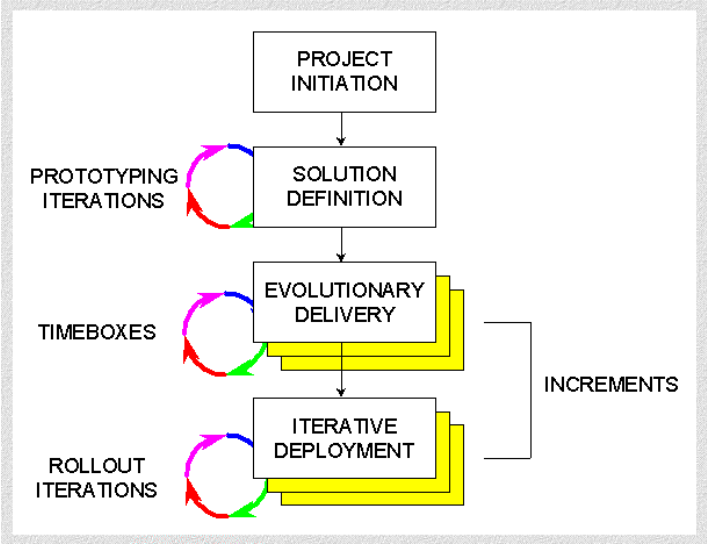
- Definição dos objetivos: estabeleça os objetivos para as atividades principais do projeto e da execução, definindo:
  - O que será construído (especificação de sistema);
  - Como se ajustará e adaptará aos processos do negócio;
  - Que restrições são conhecidas, ou serão determinadas mais adiante, na definição da sua arquitetura;
  - Que unidades incrementais de projeto serão entregues.

- Entrega evolutiva: defina múltiplos incrementos de curto ciclo de trabalho, especifique metas mais detalhadas para os incrementos e defina intervalos de

tempo para testes. Construa e teste cada unidade nos intervalos de tempo definidos.

- Distribuição iterativa: Planejamento e execução da entrega dos incrementos ao usuário do negócio, considerando: programação, hardware, infra-estrutura, treinamento, testes de aceitação, etc..

Figura 17: Catalysis – Fluxo do processo



Fonte: D'Souza (1999)

Todo esse trabalho é modelado, estruturado e concebido usando-se uma aproximação consistente em todos os estágios. Completamente separada de todo o projeto, necessita-se de uma visão voltada para programação para definir processos, arquiteturas, recursos reusáveis, padrões e estratégias para permitir um planejamento adequado de todo o projeto.

3.4.4 Distinções fundamentais

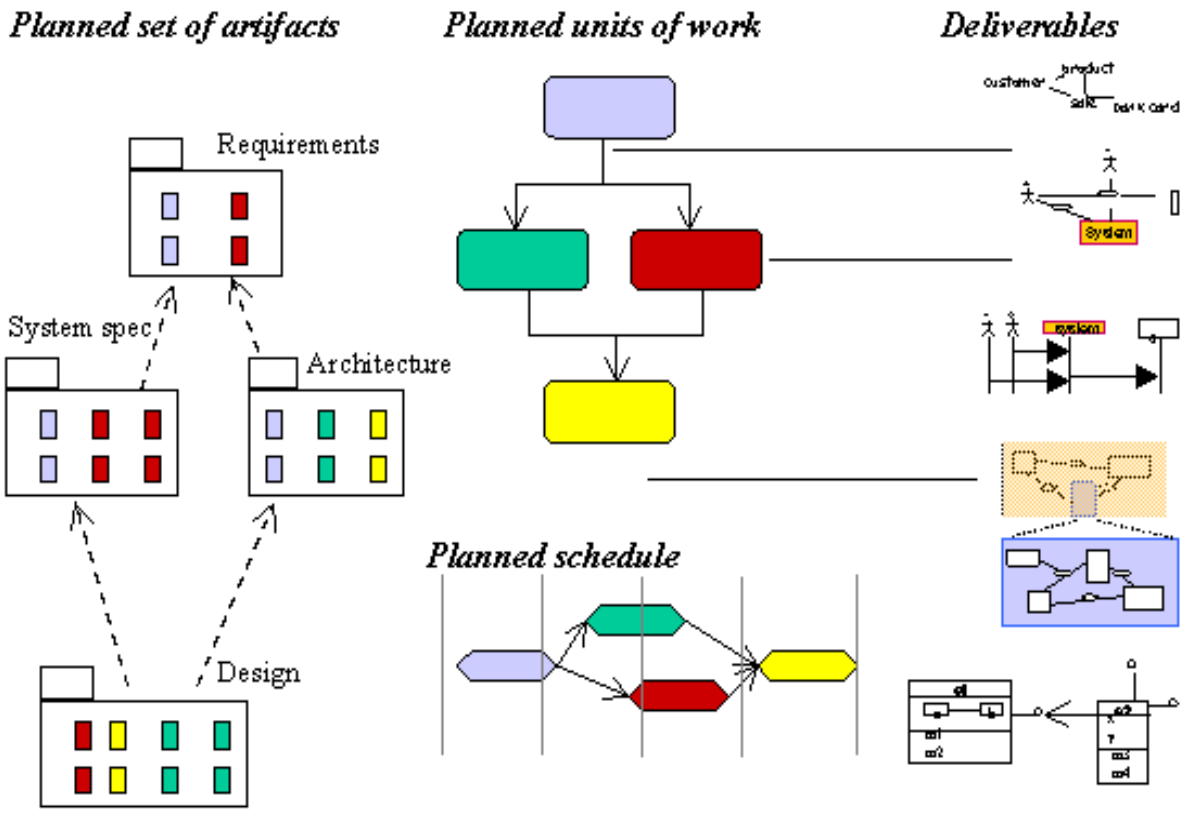
A metodologia Catalysis também faz uma distinção clara entre:

- Estrutura de artefatos*(Planned set of artifacts)*: Pode-se finalmente produzir modelos de domínio / negócio, uma especificação do sistema, o projeto e os padrões de arquitetura que são usados e a implementação. Estas partes têm um relacionamento muito bem definido, um em relação ao outro, independente do processo.
- Processo*(Planned units of work)*: o arranjo e a seqüência do trabalho para povoar a eventual estrutura de artefatos. Isso pode ser feito de forma *top-down*, *bottom-up*, *inside-out*, ou numa combinação destas formas. Este processo define

que coisas devem ser feitas e em que ordem, e as dependências entre eles. Assim, é possível ter muitas rotinas em andamento, através da mesma estrutura de artefatos, considerando cada um, processos diferentes.

- Cronograma*(Planned schedule)*: o desdobramento de um processo no tempo e outros recursos, e o progresso atual em relação ao planejado.
- Produtos liberados*(Deliverables)*: a forma concreta, modelos de documentação, diagramas que são requeridos em pontos críticos do processo. Alguns aspectos destes itens, com certeza, variaram de acordo com os processos.

Figura 18: Catalysis – Distinções fundamentais



Fonte: D'Souza (1999)

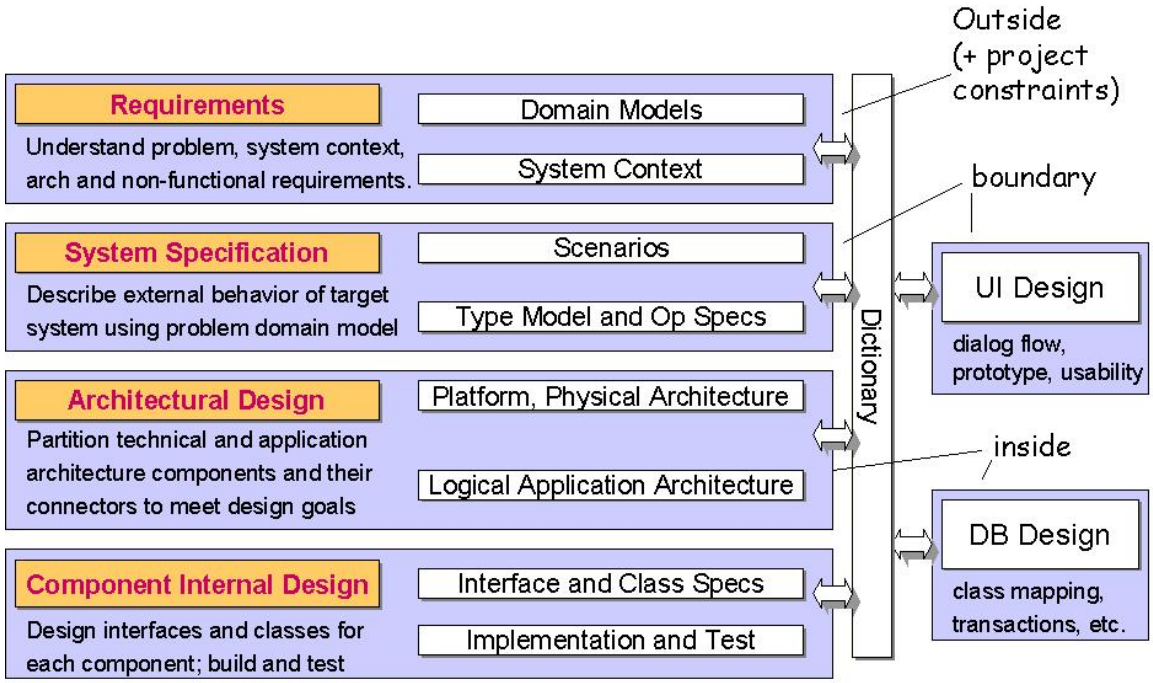
3.4.5 Perspectiva do método

A metodologia Catalysis permite a acomodação e implementação de sistemas heterogêneos envolvendo sistemas legados que interagem com aplicações novas, utilização de pacotes ERP que serão uma parte da solução, utilização de componentes de uma aplicação nova que não utiliza a tecnologia de objetos ou de

componentes e partes de um projeto que usa métodos e processos tradicionais existentes.

Possibilita uma perspectiva de utilização sistemática para analisar e ajuntar componentes heterogêneos em soluções completas. O desenvolvimento baseado em componentes da Catalysis fornece suporte adequado para projetos centrados em interfaces, fornecendo descrições precisas dos componentes. Direciona de forma clara os problemas de integração de componentes empresariais e integra projetos heterogêneos envolvendo componentes legados e orientados a objetos, dando ênfase ao reuso em soluções completas.

Figura 19: Visão geral do método Catalysis



Fonte: D'Souza (1999)

3.5 Avaliação dos métodos

Com o crescimento do uso de objetos e a redução do trabalho de programação, os desenvolvedores no futuro provavelmente serão divididos em montadores de componentes (as pessoas que desenvolvem os aplicativos) e os produtores de componentes. Componentes poderão ser adquiridos no mercado ou produzidos internamente nas organizações. Alguns se tornarão padrões de fato, enquanto que outros não (a maioria).

O desenvolvimento de aplicativos baseados em componentes não é simplesmente do tipo ligue-e-use. Muitos aspectos precisarão ser construídos, mesmo que se esteja usando componentes sofisticados.

O delineamento entre produtores e consumidores de objetos, ou componentes pré-fabricados ocorrerá, em especial, nas grandes organizações. Para um desenvolvedor independente, provavelmente ele será tanto o produtor quanto o consumidor de objetos. O programador deveria tornar-se muito mais um integrador de sistemas de softwares do que o foi no passado.

Repositórios e bibliotecas de classes cheios de todas as características imagináveis: objetos OLE, componentes ActiveX, objetos Corba, etc estarão disponíveis para utilização. O desafio é saber como entender todos esses componentes pensando em construir aplicativos com eles.

A linguagem a ser utilizada na construção desses componentes não é muito importante, porque a interoperabilidade dos objetos já é um fato, embora ainda precise de mais maturidade. As linguagens de programação estão se tornando mais especializadas em determinadas áreas, e o que vai definir a escolha de uma delas será saber quais API de comunicação lhe dão suporte.

Para se escolher uma metodologia de construção de aplicativos baseada em componentes é necessário pesar os seguintes fatores:

- O “tamanho” dos sistemas a serem desenvolvidos e o ciclo de vida esperado;
- Habilidades técnicas dos analistas e programadores da organização;
- Exigências de clientes com relação à tecnologia (conservador / moderno);
- Aderência a padrões de qualidade na organização;
- Tecnologias e metodologias em uso atualmente;
- Outros aspectos (financeiros, administrativos e tecnológicos)

Tratando-se de sistemas grande porte, o melhor é utilizar metodologias completas e rigorosas, com passos formais, revisões estruturadas, documentos intermediários formais e ciclo de vida em cascata. O desenvolvedor deverá estar interessado em reuso a longo prazo. As metodologias leves, sem muito rigor, com passos intermediários informais e ciclo de vida iterativo são mais recomendadas para sistemas menores. Documentação não é a base do projeto, mas apenas uma ferramenta de consulta. O desenvolvedor está interessado em reuso imediato.

Caso a organização esteja preocupada em adotar padrões internacionais de qualidade, deverá utilizar alguma metodologia focada no controle do processo. Quanto mais rigoroso for o método escolhido, melhor para a organização. Métodos rigorosos têm um *overlap* grande em relação às atividades de sistemas de qualidade, como SEI e CMM.

Outro fator a ser considerado é quanto às tecnologias e metodologias em uso na organização. Caso se utilize ainda de metodologias estruturadas, a equipe irá aproveitar muito do conhecimento existente ao migrar para OO utilizando a notação padrão UML. Partições básicas de modelagem (Dados, Estados e Funções) existem em OO. O Analista somente muda a ordem do desenvolvimento de cada aspecto do sistema. A evolução para a construção de componentes será uma conseqüência do amadurecimento da equipe.

4 ESTUDO DE CASO

4.1 Introdução

O presente estudo de caso tem por finalidade modelar e projetar um componente de software a partir de uma das metodologias detalhadas no capítulo anterior. O componente a ser construído é uma camada de persistência para aplicações orientadas a objeto interagirem com banco de dados relacional.

A opção pelo RUP – Rational Unified Process de desenvolvimento de *software* como metodologia, foi o fato desse processo prever um ciclo de vida de projeto iterativo e incremental característico de desenvolvimento de sistemas por prototipação, que foi usado para a implementação do componente.

Dentro da metodologia escolhida, os Modelos de Casos de Uso, de Análise e de Projeto foram escolhidos por melhor representarem as fases do ciclo de vida do *software* efetivamente realizadas, que são as fases de concepção e elaboração. Essas fases concentram-se mais sobre as atividades de levantamento de requisitos, análise e projeto (desenho). A atividade de implementação foi concluída, sendo desenvolvida a interface do usuário do protótipo, que serve para validar e capturar novos requisitos. Essa interface permite corrigir os requisitos, o mais breve possível, enquanto o custo ainda é menor que nas fases mais avançadas do ciclo de desenvolvimento.

O Modelo de Casos de Uso visa documentar os requisitos funcionais do sistema que foram capturados, servindo como um contrato entre o cliente e os desenvolvedores. Posteriormente, serve também de base para o plano de testes. Este modelo representa uma visão externa do sistema e usa a linguagem do usuário. O Modelo de Casos de Uso apresentado nesse projeto foi baseado em exemplos do capítulo três do livro de Quatrani (2000).

O Modelo de Análise define as classes de análise para realizar as funcionalidades de um caso de uso. São definidas classes fronteiriças, classes de controle e classes de entidades. Este modelo representa uma visão interna do sistema e usa a linguagem do desenvolvedor. É usado pelos desenvolvedores para entender o sistema.

O diagrama de seqüência pode fazer parte tanto de um modelo quanto de outro ou de ambos, pois essa ferramenta de modelagem permite mapear as mensagens do Modelo de Análise em operações das classes do Modelo de Projeto.

Fazem parte do Modelo de Análise os seguintes diagramas:

- Diagrama de classes de análise: representação gráfica das classes candidatas identificadas como classes que realizam os casos de uso.
- Diagrama de seqüência: representação gráfica do comportamento dinâmico e temporal dos objetos das classes de análise que colaboram para a realização dos casos de uso.
- Diagrama de estado: representação gráfica do comportamento dinâmico das instâncias das classes e dos eventos que causam mudanças de estado.
- Diagrama de classes de projeto: representação gráfica das classes de projeto com seus respectivos atributos, métodos, relacionamentos e hierarquias com as demais classes, já orientadas para suas implementações em ambiente Java.

O Modelo de Projeto descreve a realização física dos casos de uso visando um ambiente de implementação. É dependente de uma linguagem de programação e deve sofrer atualizações durante todo o ciclo de vida do *software*. O Modelo de Projeto deve tentar se ajustar o máximo possível ao modelo de análise.

Dentro do RUP, o projeto (desenho) tem seu ápice de atividade no final da fase de elaboração e início da fase de construção.

Vale a pena destacar o fato que na ferramenta CASE de modelagem *Rational Rose* o Modelo de Análise tende a se tornar ou ficar praticamente igual ao Modelo de Projeto à medida que o desenvolvimento do *software* avança além da fase de elaboração.

Abaixo, apresentamos as atividades / fases contempladas neste estudo de caso:

Quadro 10: Atividades e Fases do Estudo de Caso

Atividades	**Contemplado**	**Fases**	**Contemplado**
Workflows de Processos		1.Concepção	Sim
1.Requisitos	sim	2.Elaboração	sim
2.Análise & Projeto	sim	3.Construção	parcial
3.Implementação	sim (*)	4.Transição	não
4.Teste	sim (*)		
5.Utilização	não		

(*) Não mostrado neste trabalho

4.2 Mapeando Objetos para Banco de Dados Relacional

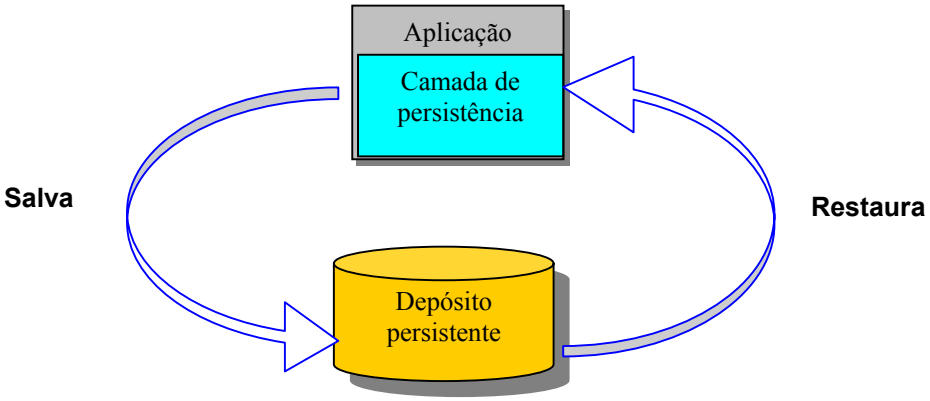
Com a chegada de aplicações que manipulam objetos, os bancos de dados foram ficando sem condições de suportar tais dados, criando-se então um problema: como fazer objetos persistirem durante todo o tempo de execução de uma aplicação, isto é, como armazená-los em algum mecanismo persistente, já que o paradigma é relacional.

Diante dessa necessidade, surgiram diversas abordagens para a implementação dos serviços e gerenciamento de objetos persistentes, utilizando bancos de dados como repositórios e suporte intrínseco de linguagens (como o JavaBeans).

Na abordagem de bancos de dados, existem duas opções:

- A primeira é a utilização de bancos de dados orientados a objetos. Esta opção teria a desvantagem de ter toda uma aplicação dependente de uma marca de bancos de dados, existindo uma grande inflexibilidade numa eventual troca do banco, pois isso acarretaria que os desenvolvedores re-escrevessem o código para o novo banco de dados, sem reaproveitamento de classes ou pacotes, uma vez que não existe um padrão de linguagem de manipulação de objetos para esses bancos, segundo Furlan(1998).
- A segunda opção seria a construção de uma camada de persistência robusta que teria a função de traduzir os objetos para bancos de dados relacionais. Qualquer banco de dados relacional poderia ser utilizado como mecanismo persistente, com a vantagem da flexibilidade de uma eventual troca do mesmo, se necessário, uma vez que, para esses bancos, existe uma linguagem padrão SQL (*Structured Query Language*) de manipulação de tabelas e dados, também de acordo com Furlan (1998).

Figura 20: Esquema de Persistência.



O objetivo deste estudo de caso é desenvolver uma camada de persistência robusta utilizando bancos de dados relacionais como repositório de dados.

4.2.1 Persistência

Quando um programa entra em execução, ele faz uso de dados. Existem casos em que se precisa que esses dados persistam além do tempo de vida de uma única execução de programa, então se utiliza um depósito permanente de dados, segundo Rumbaugh(1994).

Existem diversas razões para se desejar dados persistentes:

- Os dados persistentes fornecem o mecanismo mais fácil para passar dados de um programa para outro.
- Os dados persistentes permitem que o mesmo programa reinicie o processamento em uma data posterior.
- Depósitos de dados muitas vezes são úteis para fins de preparação de históricos ou para arquivamento.

Existem diversas abordagens para a implementação dos serviços de dados persistentes: arquivos, dispositivos de hardware, bancos de dados e suporte intrínseco de linguagens. Alguns novos bancos de dados baseados em objetos são especialmente práticos em seus serviços de armazenamentos de dados. Em grande parte, um banco de dados avançado baseado em objetos é similar a uma linguagem baseada em objetos.

Numa linguagem de programação dois tipos de objetos podem ser distinguidos: objetos temporários e objetos persistentes. Um **objeto temporário** é um objeto que perde o valor quando o processo termina. Exemplos típicos são as variáveis automáticas – quando as variáveis ficam fora do escopo, são destruídas, e seu valor é perdido. Um **objeto persistente** é um objeto que pode ter seu valor restaurado após o processo ter terminado a sua execução.

4.2.2 Persistência utilizando banco de dados relacional

Segundo Ambler(1999), a fase de implementação é uma das partes mais complexas no projeto orientado a objetos. Todos os detalhes da análise podem se perder quando os modelos são transformados em software sem o devido cuidado. A utilização de gerenciadores de bancos de dados relacionais para armazenar grandes

volumes de informação é comum a muitas aplicações, entretanto, com a introdução da tecnologia de objetos os programadores são colocados em um impasse, porque os bancos atuais não oferecem uma boa aderência aos princípios da orientação a objetos. Pode-se construir com sucesso um modelo de objetos, mas exige-se cautela se as ferramentas de implementação não são adequadas para representá-lo completamente.

Os objetos formam uma unidade que encapsula atributos e operações. Os bancos de dados relacionais são eficientes para representar os atributos, mas são muito limitados quanto às operações. Pode-se tentar estabelecer uma analogia direta entre classes de objetos e tabelas, e entre atributos e colunas. Entretanto, alguns aspectos merecem destaque nesta analogia, para garantir que todas as características do modelo de objetos sejam reproduzidas com precisão pelo banco de dados relacional.

A idéia geral é partir do modelo obtido pela análise orientada a objetos, e fazê-lo passar por um processo de tradução dos seus elementos em tabelas e colunas. Este processo é executado em duas etapas:

- Seleção dos objetos persistentes: A primeira etapa da tradução é caracterizar as informações persistentes no modelo de objetos. Identificadas as informações persistentes e que devem ser mantidas em um banco de dados, é conveniente isolá-las em objetos persistentes. Deste modo, forma-se uma camada de objetos persistentes no modelo.
- Tradução dos objetos e relacionamentos em tabelas: Inicia-se a tradução criando uma tabela para cada objeto persistente, sendo que cada atributo dá origem a uma coluna. É conveniente adotar como nome da tabela o mesmo nome do objeto para aumentar a legibilidade do software. Deve-se manter os atributos entre os tipos básicos encontrados nos bancos de dados, como: número, caractere, datas, etc. Os tipos complexos devem ser tratados em separado, através de objetos e associações auxiliares. Escolhe-se uma chave primária para cada tabela. A tradução segue agora para as associações e agregações que ocorrem entre os objetos persistentes. A tradução de uma herança entre os objetos persistentes necessita de esforço maior de abstração, uma vez que os bancos de dados normalmente não dão suporte a este tipo de relacionamento. Após a tradução é impossível dizer quem herda os atributos de quem, ficando este controle fora da estrutura do banco de dados. Cria-se uma tabela para a

classe-mãe e uma para cada classe-filha, incluindo nas tabelas das classes - filhas, além dos seus próprios atributos, uma coluna para a chave primária da classe-mãe.

Precisa-se designar um identificador único para um objeto, pelo qual se possa identificá-lo. Na terminologia relacional um identificador único é chamado de chave ou chave primária. Na terminologia de objetos ele é chamado de identificador de objeto (OID - *object identifier*). OIDs são tipicamente implementados como objetos em uma aplicação orientada a objetos e podem ser mapeados para uma coluna da tabela.

4.2.3 Questões básicas do mapeamento

O mapeamento de objetos para bancos de dados relacionais envolve os seguintes tópicos:

- Mapear atributos para colunas - Um atributo de uma classe será mapeado para zero ou mais colunas num banco de dados relacional. Um ponto importante é que nem todos os atributos são persistentes.
- Mapear classes para tabelas
- Implementando hierarquia em um banco de dados relacional
    - Mapear diversas classes para uma simples tabela
- Mapear relacionamentos:
    - Um-para-um
    - Um-para-muitos
    - Muitos-para-muitos
    - Associações e Agregações
    - Mesmas classes/tabelas, diferentes relacionamentos

Exceto para bancos de dados muito simples, raramente existe um mapeamento um-para-um de classes para tabelas. Quando se mapeia objetos para bancos de dados relacionais existem duas situações onde faz sentido usar procedimentos armazenados (*stored procedures*). Primeiro, quando se está construindo um protótipo, assumindo que não se tenha uma camada de persistência sólida já construída, então este é provavelmente o caminho mais rápido para gerar o protótipo e executá-lo. Segundo, quando se está mapeando para um banco de dados, cujo projeto é completamente não apropriado para objetos e não é possível

refazer as necessidades específicas, pode-se criar procedimentos armazenados para ler e escrever registros que manipulem os objetos necessários.

Existem, entretanto, várias razões para não usar procedimentos armazenados quando se mapeia objetos para bancos de dados relacionais. Primeiro, o servidor pode rapidamente se tornar um gargalo usando esta aproximação. Segundo, *stored procedures* são escritos em uma linguagem proprietária, isso causa uma complicação quando é necessário mudar o banco de dados. Uma outra coisa a ser ponderada é que a indústria está mudando, devendo-se considerar pelo menos uma atualização no banco de dados. Terceiro, pode-se incrementar dramaticamente o acoplamento dentro do banco de dados porque procedimentos armazenados acessam diretamente tabelas, acoplando as tabelas para procedimentos armazenados. Este incremento no acoplamento, reduz a flexibilidade dos administradores de banco de dados, quando da reorganização do banco de dados. Os administradores de banco de dados precisam reescrever os procedimentos armazenados e aumentar a responsabilidade dos desenvolvedores porque eles têm que tratar com o código dos procedimentos armazenados.

O paradigma dos objetos é diferente do paradigma relacional, mas em 99% dos casos, o ambiente de desenvolvimento é orientado a objetos e o mecanismo de persistência é um banco de dados relacional, segundo Ambler(1998).

Uma camada de persistência encapsula acessos para bancos de dados, permitindo a programadores de aplicações se aplicarem nos seus problemas. Isto significa que as classes de acesso a bancos de dados são encapsuladas provendo uma interface simples e completa para os programadores das aplicações. Futuramente, o projeto dos bancos de dados será encapsulado para que programadores não precisem conhecer os detalhes do formato do banco de dados. Uma camada de persistência encapsula completamente os mecanismos de armazenamento permanente, protegendo o programador das mudanças.

Uma implicação é que a camada de persistência precisa usar um dicionário de dados que forneça a informação necessária para mapear objetos para tabelas. Quando o domínio de negócios mudar, e isto sempre acontece, não é necessário mudar algum código na camada de persistência, é necessário mudar apenas o dicionário de dados.

Construir e manter uma camada de persistência é uma tarefa muito complexa. Para começar o desenvolvimento de uma camada de persistência, é necessário que

o desenvolvedor tenha a certeza de que pode terminar a tarefa, pois ela inclui o suporte e a manutenção da camada.

4.3 Definição do problema

Antes de dar início à primeira fase da modelagem, faremos algumas considerações sobre o que a camada se propõe a fazer e outras observações importantes para o bom entendimento do problema.

A primeira etapa a ser feita em um desenvolvimento de software é definir os requisitos para o software. Os requisitos aqui apresentados refletem experiências de anos de construção de camadas de persistência de Ambler (1997, 1998, 1999).

Uma camada de persistência encapsula o comportamento necessário para fazer objetos persistirem, ou em outras palavras ler, escrever e apagar objetos em mecanismos de persistência. Uma camada de persistência deve suportar:

- Completo encapsulamento do mecanismo persistente **-** O ideal é ter somente que enviar mensagens *save, delete e retrieve* para um objeto ser salvo, eliminado ou recuperado e a camada cuida do resto.
- Ações envolvendo muitos objetos - É comum recuperar e excluir diversos objetos de uma só vez, então uma camada de persistência robusta tem que ser capaz de suportar a recuperação e exclusão de muitos objetos simultaneamente.
- Transações - Uma camada de persistência robusta tem que ser capaz de suportar transações, que é uma coleção de ações em diversos objetos. Uma transação poderia ser feita de alguma combinação de operações como salvar, recuperar e eliminar objetos. Transações podem ser planas, uma aproximação tudo ou nada, onde todas as ações podem ser bem sucedidas ou canceladas. Transações podem também ser de curta duração, executando em centésimos de segundo ou de longa duração, demorando horas para terminar.
- Extensibilidade - Deve ser capaz de adicionar novas classes para as aplicações e ser capaz de mudar o mecanismo de persistência facilmente. Em outras palavras, a camada de persistência deve ser flexível suficiente para permitir que programadores das aplicações e administradores do mecanismo persistente possam fazer o que eles precisam.
- Identificadores de objetos - Um identificador de objetos ou OID, é um atributo, tipicamente um número, que unicamente identifica um objeto. OIDs são valores

orientados a objetos, equivalentes às chaves da teoria relacional, que unicamente identificam uma linha dentro de uma tabela.

- Cursores - Uma camada de persistência que suporta a habilidade de recuperar muitos objetos com um simples comando, também suportaria a habilidade para recuperar mais de um objeto. Um conceito interessante do mundo relacional é o de cursor. Um cursor é uma conexão lógica para um banco de dados do qual pode-se recuperar objetos usando uma aproximação controlada. Esta aproximação é freqüentemente mais eficiente que retornar centenas ou milhares de objetos todos de uma só vez, porque muitos usuários não necessitam de todos os objetos imediatamente.
- Registros - Uma grande maioria de ferramentas de geração de relatórios pega uma coleção de registros como entrada, não coleções de objetos. Se uma empresa usa semelhante ferramenta para gerar relatórios dentro de uma aplicação orientada a objetos, então a camada de persistência precisa suportar a habilidade para simplesmente retornar registros como resultado de uma requisição de recuperação de modo a não ter uma sobrecarga adicional para converter os registros do banco de dados para objetos e vice-versa.
- Arquiteturas múltiplas - Empresas mudam de arquiteturas centralizadas para arquiteturas cliente/servidor de duas camadas, ou para arquiteturas com n camadas, ou para ambientes distribuídos. Então a camada precisa ser capaz de suportar essas diversas aproximações.
- Várias versões e/ou marcas de bancos de dados - Uma camada de persistência deve suportar a habilidade para facilmente mudar de banco de dados sem ter que alterar o seu código fonte. Portanto, uma ampla variedade de versões e marcas de bancos de dados poderia ser suportada pela camada de persistência.
- *Drivers* nativos e não nativos **-** Existem diversas estratégias diferentes para acessar um banco de dados relacional e uma boa camada de persistência suportará as estratégias mais comuns. Estratégias de conexão incluem usar *Open Database Conectivity* (ODBC), *Java Database Conectivity* (JDBC), e *drivers* nativos fornecidos por marcas de bancos de dados.
- Consultas SQL (*Structured Query Language*) - Escrever chamadas SQL no código orientado a objetos viola o encapsulamento, pois assim acopla-se a aplicação diretamente no banco de dados. Às vezes, por razões de desempenho

precisa-se fazer isto, mas esta situação deve ser uma exceção e não uma norma, uma exceção que precisa ser bem justificada antes de ocorrer.

Camadas de persistência devem permitir que desenvolvedores de aplicações se concentrem no que eles são melhores, que é desenvolver aplicações, sem ter a preocupação de como seus objetos serão armazenados.

Futuramente, camadas de persistência também permitirão a administradores de bancos de dados (DBAs) administrar bancos de dados, sem ter preocupações sobre acidentalmente introduzir erros dentro das aplicações existentes. Com uma camada de persistência bem construída, DBAs serão capazes de mover tabelas, renomear tabelas, renomear colunas e reorganizar tabelas sem ter que alterar o código fonte das aplicações que acessam estas tabelas e colunas.

4.3.1 Modelagem do domínio

4.3.1.1 Requisitos

Nesta etapa inicial são levantados os atores, os requisitos funcionais representados pelos casos de uso e requisitos não funcionais do sistema. Os atores são os usuários, dispositivos (máquinas) e outros processos que interagem com o sistema.

Os atores e os casos de uso, bem como seus inter-relacionamentos são representados através de diagramas de casos de uso. Cada um desses diagramas representará a visão do sistema por um de seus atores, no caso deste estudo de caso um ator.

4.3.1.2 Atores

Os atores representam os tipos de usuários que interagem com o sistema, aqueles que literalmente o utilizam. A camada de persistência terá apenas um ator, o desenvolvedor. O desenvolvedor será o único usuário a interagir com a camada de persistência.

O desenvolvedor terá a função de importar o pacote que contém todas as classes da camada de persistência e a partir dele, utilizar todas as funcionalidades

do sistema, tais como salvar, recuperar, atualizar e excluir um objeto ou ainda recuperar e excluir coleções de objetos baseados em algum critério.

4.3.1.3 Casos de uso (requisitos funcionais)

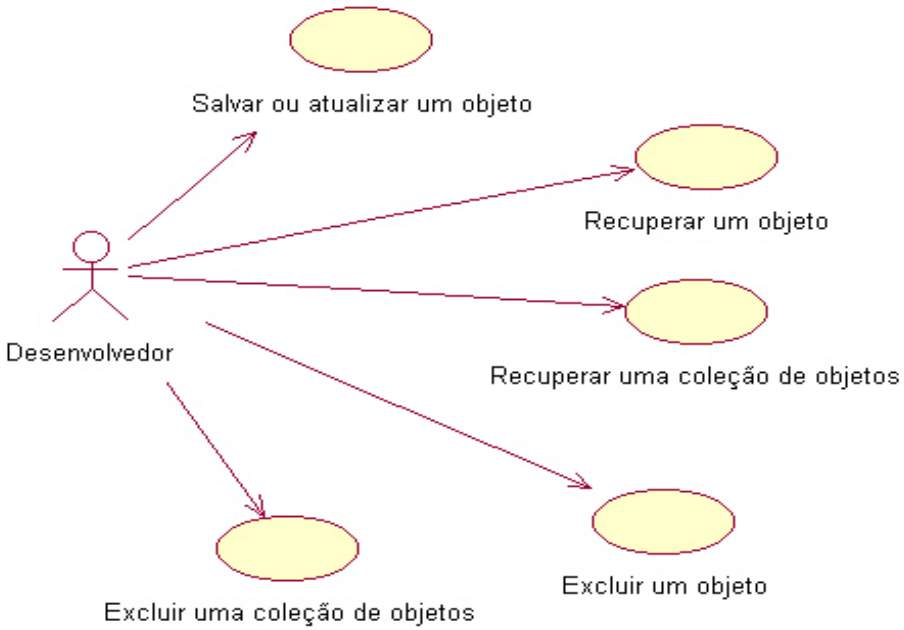
De acordo com a proposta, a camada de persistência implementará as funções básicas que serão desempenhadas pelo desenvolvedor de aplicações orientadas a objetos.

Os casos de uso da camada de persistência são:

- Salvar ou atualizar um objeto para um mecanismo persistente.
- Recuperar um objeto de um mecanismo persistente.
- Excluir um objeto de um mecanismo persistente.
- Recuperar uma coleção de objetos de um mecanismo persistente.
- Excluir uma coleção de objetos de um mecanismo persistente.

A seguir será descrito o diagrama de casos de uso para a camada de persistência:

Figura 21: Diagrama de casos de uso.



Para cada caso de uso da camada de persistência será dada uma especificação para o fluxo de dados

Caso de Uso 1:

Cenário 1: Salvar ou atualizar um objeto para um mecanismo persistente

O desenvolvedor utilizará o comando “save” tanto para o caso de salvar um objeto, quanto para o caso de atualizar um objeto no banco de dados. A camada terá a tarefa de saber se o objeto persistente já existe ou não no banco de dados, para saber então se vai criar uma expressão SQL para inserção ou para atualização

Fluxo principal:

1 Inicializa-se uma conexão persistente com o banco de dados e se verifica se existe um banco de dados conectado.

2 Pega-se o dicionário de classes a partir da classe do objeto a ser salvo ou atualizado.

3 Pega-se o nome do banco de dados e então cria-se uma conexão persistente para aquele banco de dados.

4 Verifica se o objeto a ser salvo é novo. Se for novo, executa passos 5 e 6 para salvar. Se não for novo, executa passos 7 e 8 para atualizar.

5 Monta uma expressão SQL para inserção pegando o correspondente nome da tabela para aquele objeto a ser salvo.

6 Executa a expressão SQL montada para inserir objeto e vai para passo 9.

7 Monta uma expressão SQL para atualização pegando o correspondente nome da tabela para aquele objeto a ser recuperado.

8 Executa a expressão SQL montada para atualizar o objeto no banco de dados.

9 Fecha a conexão persistente.

10 Libera a conexão persistente ao banco de dados.

Exceções:

3.a O banco de dados não existe.

Tratamento de erro:

3.a Avisa o desenvolvedor de que o banco de dados não existe.

Caso de Uso 1:

Cenário 2: Salvar ou atualizar um objeto para um mecanismo persistente fazendo parte de uma transação

O caso de uso “salvar ou atualizar um objeto” pode ser executado também através de uma transação, assim o comando para salvar ou atualizar um objeto fará parte de um conjunto de operações, que podem suceder ou falhar como um todo.

Fluxo principal:

1 Inicializa-se a classe de transação.

2 Inicializa-se uma conexão persistente para o banco de dados e se verifica se existe um banco de dados conectado.

3 Pega-se o dicionário de classes a partir da classe do objeto a ser salvo ou atualizado.

4 Inicializa-se uma nova transação.

5 Acrescenta um objeto em uma transação para ser salvo.

6 Pega-se o nome do banco de dados e então cria-se uma conexão persistente para aquele banco de dados.

7 Associa a transação ao banco de dados.

8 Verifica se o objeto a ser salvo é novo. Se for novo, executa passos 9 e 10 para salvar. Se não for novo, executa passos 11 e 12 para atualizar.

9 Monta uma expressão SQL para inserção pegando o correspondente nome da tabela para aquele objeto a ser salvo.

10 Executa a expressão SQL montada para inserir o objeto e vai para o passo 13.

11 Monta uma expressão SQL para atualização pegando o correspondente nome da tabela para aquele objeto a ser recuperado.

12 Executa a expressão SQL montada para atualizar o objeto no banco de dados.

13 Fecha a conexão persistente.

14 Libera a conexão persistente ao banco de dados.

Exceções:

3.a Lista de banco de dados vazia

5.a Transação não iniciada.

Tratamento de erro:

3.a Avisa o desenvolvedor de que não existe banco de dados disponível.

5.a Avisa o desenvolvedor de que a transação não pode ser iniciada.

Caso de Uso 2: Recuperar um objeto de um mecanismo persistente

O desenvolvedor utilizará sempre o comando *retrieve* para o caso de recuperar um objeto. Sem usar um critério de recuperação, usa-se apenas o identificador do objeto para recuperá-lo.

Fluxo principal:

1 Pega-se o nome da classe do objeto a ser recuperado.
2 Pega-se o dicionário de classes a partir da classe do objeto a ser recuperado.
3 Inicializa-se uma conexão persistente.
4 Pega o nome da tabela correspondente ao objeto a ser recuperado e monta uma expressão SQL para um *select* naquela tabela.
5 Pega-se o nome do banco de dados e então cria-se uma conexão para aquele banco de dados.
6 Pega-se uma instância da conexão para o banco de dados.
7 Executa a consulta a uma tabela do banco de dados.
8 Fecha a conexão persistente.
9 Libera a conexão persistente ao banco de dados.

Exceções:

3.a Lista de banco de dados vazia.

5.a Lista de banco de dados vazia.

Tratamento de erro:

3.a Avisa o desenvolvedor de que a conexão persistente não pode ser inicializada.

5.a Avisa o desenvolvedor de que não existe banco de dados na lista de bancos de dados.

Caso de Uso 3: Recuperar uma coleção de objetos de um mecanismo persistente

O desenvolvedor utilizará também o comando *retrieve* para o caso de recuperar uma coleção de objetos, mas para isso terá que definir um critério de seleção.

Fluxo principal:

1 Pega-se o nome da classe da coleção de objetos a serem recuperados.
2 Inicializa-se a classe de critério e define-se o critério para recuperação
3 Pega-se o dicionário de classes a partir da classe do objeto a ser recuperado.
4 Pega o nome da tabela correspondente ao objeto salvo e monta uma básica expressão SQL para um *select* naquela tabela.
5 Inicializa-se a classe que suportará o conjunto retornado pela consulta.
6 Inicializa-se uma conexão persistente.
7 Pega-se o nome do banco de dados e então cria-se uma conexão para aquele banco de dados.
8 Constrói-se uma completa expressão SQL de recuperação baseada no critério escolhido.
9 Executa a consulta a uma tabela do banco de dados.
10 Fecha a conexão persistente.
11 Libera a conexão persistente ao banco de dados.

Exceções:

6.a Lista de banco de dados vazia.

7.a Lista de banco de dados vazia.

Tratamento de erro:

6.a Avisa o desenvolvedor de que a conexão persistente não pode ser inicializada.

7.a Avisa o desenvolvedor que não existe banco de dados na lista de bancos de dados.

Caso de Uso 4:

Cenário 1: Excluir um objeto de um mecanismo persistente

O desenvolvedor deverá sempre usar o comando *delete* quando quer simplesmente excluir um objeto do banco de dados. Sem utilizar um critério de exclusão, um objeto é excluído utilizando o seu identificador.

Fluxo principal:

1 Inicializa-se uma conexão persistente.

2 Pega-se o dicionário de classes a partir da classe do objeto a ser excluído.

3 Cria-se uma conexão persistente para este banco de dados.
4 Monta uma expressão de um *delete* para um objeto de uma dada tabela do banco.
5 Executa a expressão *delete* em uma tabela do banco de dados.
6 Fecha a conexão persistente.
7 Libera a conexão persistente.

Exceções:
1.a Lista de banco de dados vazia.
3.a Lista de banco de dados vazia.

Tratamento de erro:
1.a Avisa o desenvolvedor de que a conexão persistente não pode ser inicializada.
3.a Avisa o desenvolvedor de que não existe banco de dados na lista de bancos de dados.

Caso de Uso 4:
Cenário 2: Excluir um objeto de um mecanismo persistente fazendo parte de uma transação

O caso de uso de excluir um objeto pode ser executado também através de uma transação, assim o comando para excluir um objeto fará parte de um conjunto de operações que pode suceder ou falhar como um todo.

Fluxo principal:
1 Inicializa-se a classe de transação.
2 Inicializa-se a conexão persistente para o banco de dados, e verifica se existe um banco de dados conectado.
3 Pega-se o dicionário de classes a partir da classe do objeto a ser excluído.
4 Inicia-se uma nova transação.
5 Acrescenta o objeto na unidade de trabalho para ser excluído.
6 Pega-se o nome do banco de dados e então cria-se uma conexão persistente para aquele banco de dados.
7 Associa a unidade de trabalho ao banco de dados.
8 Monta uma expressão de um *delete* para um objeto de uma dada tabela do banco.
9 Executa o *delete* em uma tabela do banco de dados.

10 Fecha a conexão persistente.
11 Libera a conexão persistente.
12 Confirma a transação.

Exceções:
3.a Lista de banco de dados vazia
5.a Transação não iniciada.

Tratamento de erro:
3.a Avisa o desenvolvedor de que não existe banco de dados disponível.
5.a Avisa o desenvolvedor de que a transação não pode ser iniciada.

Caso de Uso 5:
Cenário 1: Excluir uma coleção de objetos de um mecanismo persistente

O desenvolvedor utilizará também o comando *delete* para o caso de excluir uma coleção de objetos, mas para isso terá que definir um critério de exclusão.

Fluxo principal:
1 Pega-se o nome da classe da coleção de objetos a serem excluídos e o critério de exclusão.
2 Inicializa-se uma conexão persistente.
3 Pega-se o dicionário de classes a partir da classe do objeto a ser excluído.
4 Pega-se o nome do banco de dados e então cria-se uma conexão para aquele banco de dados.
5 Pega o nome da tabela correspondente ao objeto salvo e monta uma básica expressão SQL para um *delete* naquela tabela.
6 Pega-se uma conexão persistente para o banco de dados.
7 Constrói-se uma completa expressão SQL de exclusão baseada no critério escolhido.
8 Executa a exclusão a uma tabela do banco de dados.
9 Fecha a conexão persistente.
10 Libera a conexão persistente ao banco de dados.

Exceções:

2.a Lista de banco de dados vazia.

4.a Lista de banco de dados vazia.

Tratamento de erro:

2.a Avisa o desenvolvedor de que a conexão persistente não pode ser inicializada.

4.a Avisa o desenvolvedor de que não existe banco de dados na lista de bancos de dados.

Caso de Uso 5:

Cenário 2: Excluir uma coleção de objetos de um mecanismo persistente fazendo parte de uma transação.

Esta operação poderá fazer parte também de uma transação. Para isso o desenvolvedor utilizará também o comando *delete* para o caso de excluir uma coleção de objetos, mas para isso terá que definir um critério de exclusão.

Fluxo principal:

1 Inicializa-se a classe de transação.

2 Inicializa-se a conexão persistente para o banco de dados, e verifica se existe um banco de dados conectado.

3 Pega-se o dicionário de classes a partir da classe do objeto a ser excluído.

4 Inicia-se uma nova transação.

5 Acrescenta o objeto na unidade de trabalho para ser excluído.

6 Pega-se o nome do banco de dados e então cria-se uma conexão persistente para aquele banco de dados.

7 Associa a unidade de trabalho ao banco de dados.

8 Pega o nome da tabela correspondente ao objeto salvo e monta uma básica expressão SQL para um *delete* naquela tabela.

9 Pega-se uma conexão persistente para o banco de dados.

10 Constrói-se uma completa expressão SQL de exclusão baseada no critério escolhido.

11 Executa a exclusão a uma tabela do banco de dados.

12 Fecha a conexão persistente.

13 Libera a conexão persistente ao banco de dados.
14 Confirma transação.

Exceções:
3.a Lista de banco de dados vazia.
5.a Transação não iniciada.

Tratamento de erro:
3.a Avisa o desenvolvedor de não existe banco de dados disponível.
5.a Avisa o desenvolvedor de que a transação não pode ser iniciada.

4.3.1.4 Requisitos não funcionais

- O desenvolvedor deve utilizar a camada de persistência localmente e não remotamente, pois não foi definido o mecanismo de geração de OIDs.
- O desenvolvedor deverá desenvolver as suas aplicações utilizando a linguagem de programação Java para poder fazer uso da camada de persistência.
- A camada de persistência não permite acesso simultâneo de mais de um desenvolvedor.
- O banco de dados a ser utilizado como mecanismo persistente tem que ser um banco de dados relacional que utiliza o padrão SQL.
- Todo o projeto da camada de persistência será implementado utilizando o sistema gerenciador de bancos de dados(SGBD) Oracle 8i Personal.
- Todo o projeto deve ser modelado na linguagem UML(Unified Modeling Language).

4.4 Análise

Na fase de análise, utiliza-se o diagrama de casos de uso para definir o diagrama de classes do sistema. Este primeiro diagrama da fase de análise, não deverá analisar quaisquer aspectos ligados à implementação do sistema.

4.4.1 Responsabilidades das classes

Classe ConnectionBroker - Mantém conexões para o mecanismo persistente(neste projeto um banco de dados relacional) e gerencia a comunicação entre a aplicação orientada a objetos e o mecanismo persistente.

Classe PersistentConnection - Encapsula o comportamento necessário para criar uma conexão para bancos de dados relacionais.

Classe PersistentObject - Encapsula o comportamento necessário para poder criar instâncias de objetos persistentes, podendo assim, um objeto persistente ser salvo, recuperado, atualizado ou excluído de um banco de dados relacional. Esta classe sabe como armazenar seus atributos dando uma linha de resultados de um banco de dados.

Classe PersistentSource - É a classe que permite ao desenvolvedor ter acesso para os fundamentais comandos para fazer objetos persistirem, comandos como *save, delete e retrieve*.

Classe SQLStatement - Esta hierarquia de classes sabe como construir expressões *insert, update, delete e select* no padrão SQL(*structured query language*).

Classe Trans - Permite que um conjunto de operações em objetos sejam agrupadas em uma transação que deve suceder ou falhar como um todo.

Classe SelectionCriteria - Esta hierarquia de classes encapsula o comportamento necessário para criar expressões SQL para recuperar, atualizar ou excluir coleções de objetos baseados em algum critério.

Classe PersistentCriteria - É usada para definir os critérios a serem construídos utilizando a hierarquia de classes SelectionCriteria.

Classe PersistentSet - Esta classe encapsula o conceito de cursor de banco de dados.

Classe ClassDictionary - Encapsula o comportamento necessário para mapear atributos de uma classe em colunas de uma tabela de um banco de dados relacional.

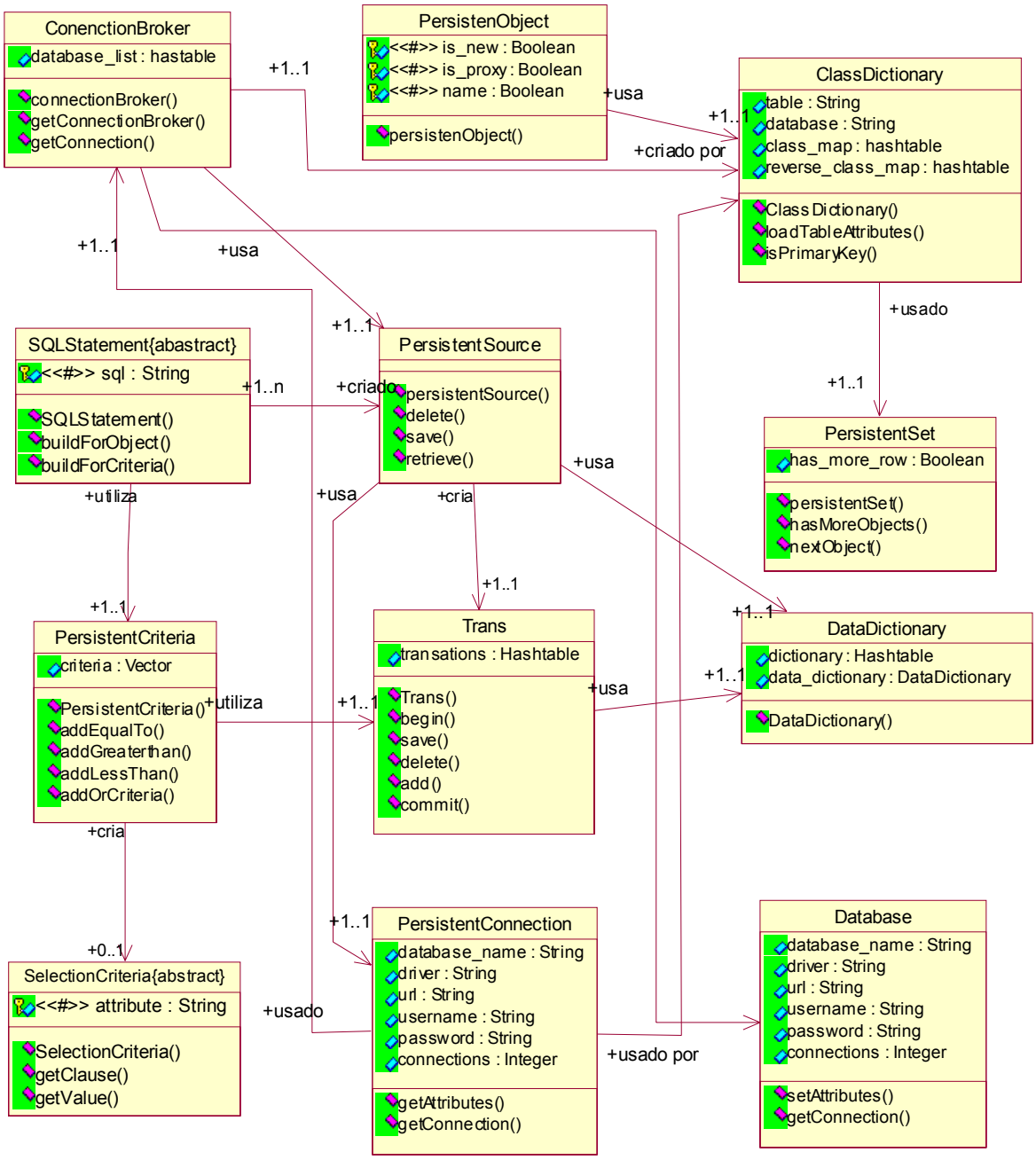
Classe DataDictionary - Vai ser responsável por manter uma correspondência entre classes e tabelas em um banco de dados, esta classe vai saber qual é nome de uma tabela que representa uma classe no banco de dados relacional.

Classe Database - Encapsula informações sobre um banco de dados, informações utilizadas na criação de uma conexão utilizando JDBC. Implementará comandos para gerenciamento das conexões ao banco de dados.

4.4.2 Diagrama de classes

O domínio de classes da camada de persistência é: ConnectionBroker, PersistentConnection, PersistentObject, PersistentSource, SQLStatement, Trans, SelectionCriteria, PersistentCriteria, PersistentSet, ClassDictionary, DataDictionary, Database.

Figura 22: Diagrama de classes (fase de análise).



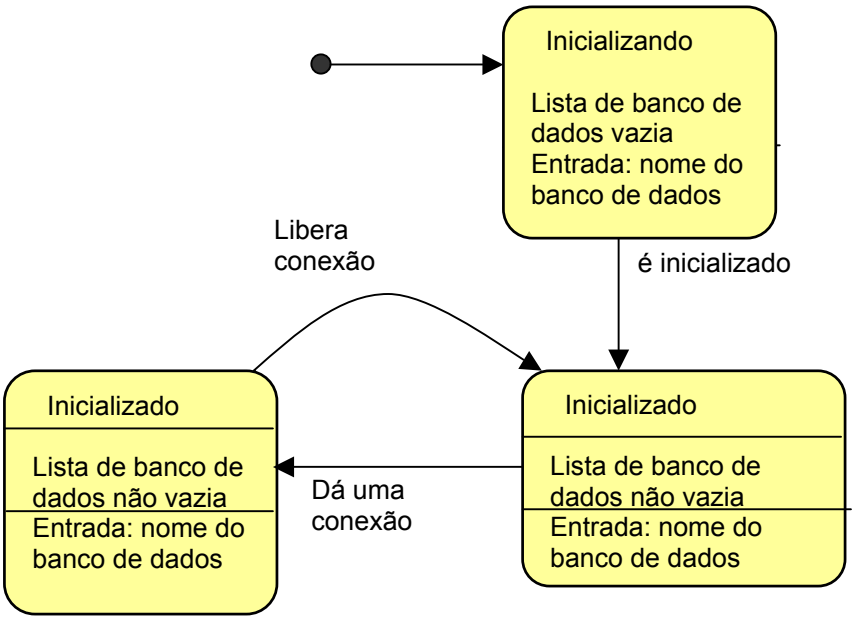
4.4.3 Diagramas de estado

Para descrever o comportamento dos objetos das classes, utilizam-se os diagramas de estados. O diagrama de estados mostra os diferentes estados que os objetos destas classes podem ter, junto com os eventos que causam essas mudanças. Serão feitos os diagramas de estado para as classes que compõem o domínio do problema.

Classe ConnectionBroker

Nesta classe um objeto terá a principal função de distribuir conexões ao banco de dados, bem como liberar conexões, ou seja, terá a responsabilidade do gerenciamento das conexões.

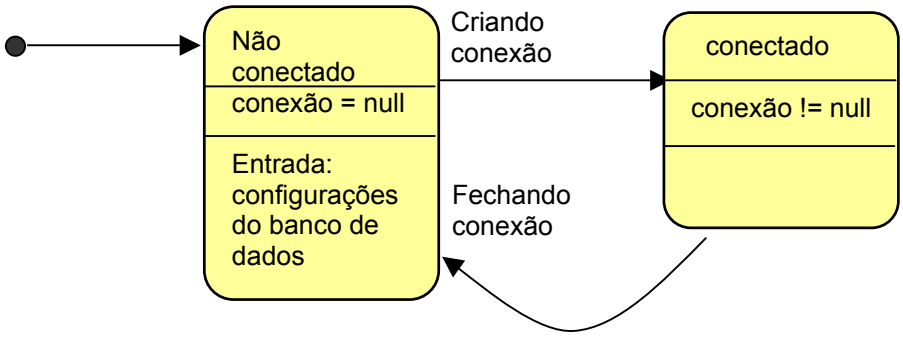
Figura 23: Diagrama de estados para a classe ConnectionBroker.



Classe PersistentConnection

Um objeto do tipo PersistentConnection terá a única função de criar uma conexão e fechar uma conexão, pois o gerenciamento da conexão é função da classe ConnectionBroker.

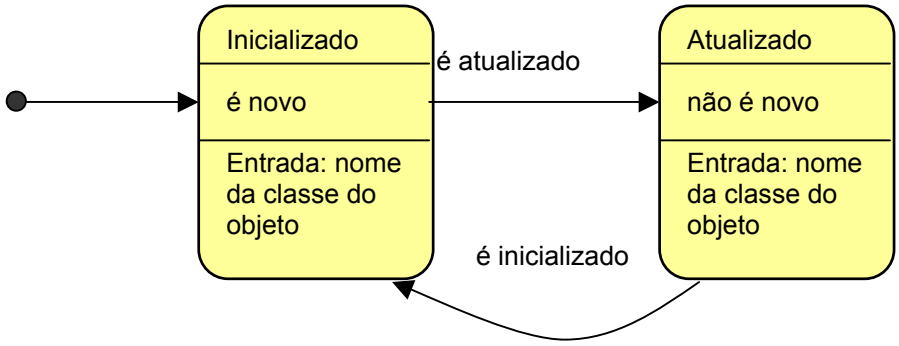
Figura 24: Diagrama de estados para a classe PersistentConnection.



Classe PersistentObject

Um objeto armazenará as informações sobre o objeto persistente, informações importantes para fazer o objeto persistir no banco de dados relacional.

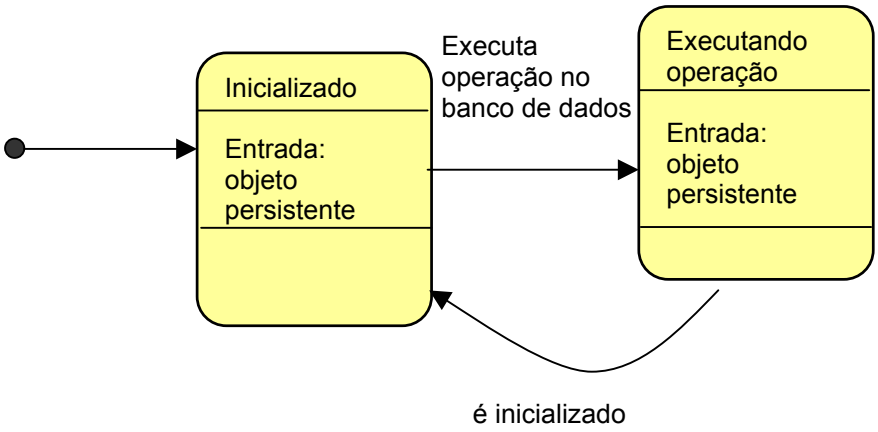
Figura 25: Diagrama de estados para a classe PersistentObject.



Classe PersistentSource

Através dos objetos do tipo PersistentSource o desenvolvedor terá acesso aos principais métodos que fazem objetos persistirem.

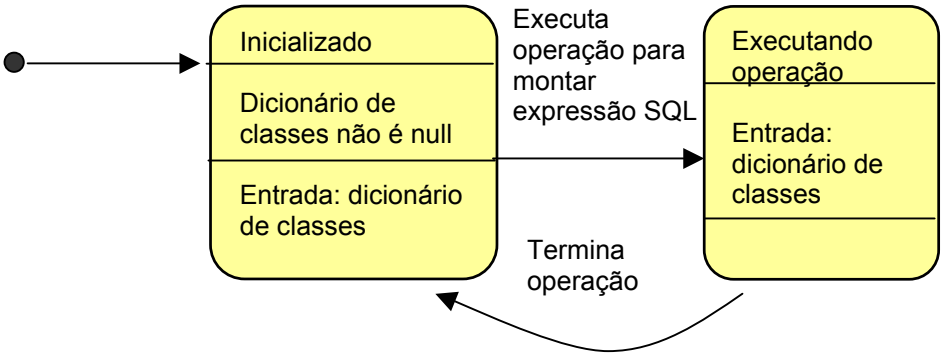
Figura 26: Diagrama de estados para a classe PersistentSource



Classe SQLStatement

Nesta classe o objeto servirá para definir comandos para criar os diferentes tipos de expressões SQL de acesso ao banco de dados.

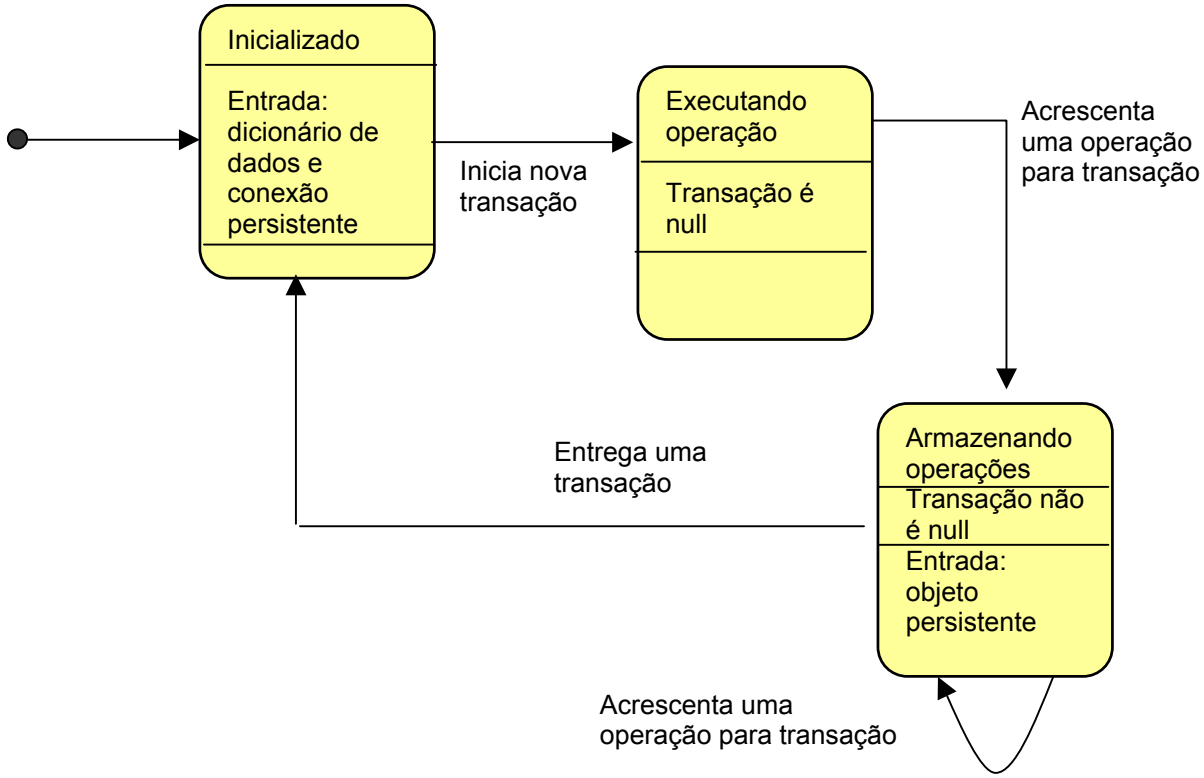
Figura 27: Diagrama de estados para a classe SQLStatement.



Classe Trans

Os objetos desta classe forneceram acesso aos principais passos de uma transação em um banco de dados, que serão utilizados pelo desenvolvedor.

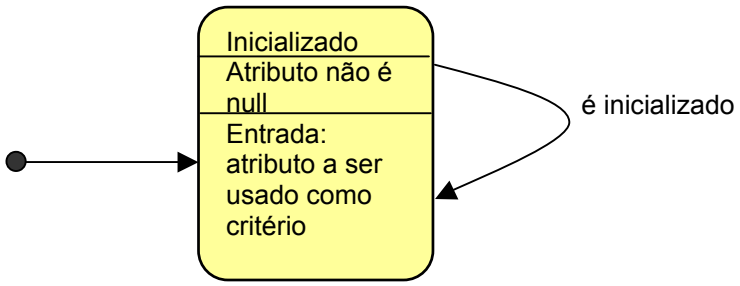
Figura 28: Diagrama de estados para a classe Trans.



Classe SelectionCriteria

A classe SelectionCriteria representa uma hierarquia de classes, que irá fornecer mecanismos para criar cláusulas no padrão SQL, que definirão os critérios.

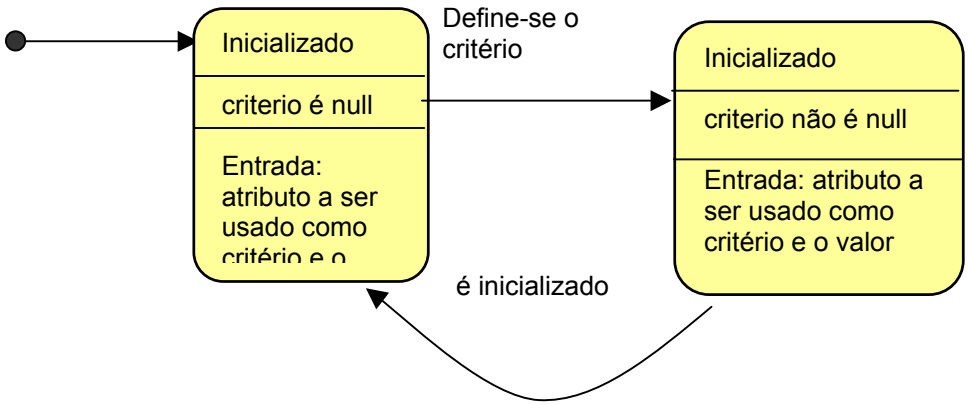
Figura 29: Diagrama de estados para a classe SelectionCriteria.



Classe PersistentCriteria

Os objetos desta classe fornecerão métodos de interface para o desenvolvedor, para que este possa definir o critério desejado.

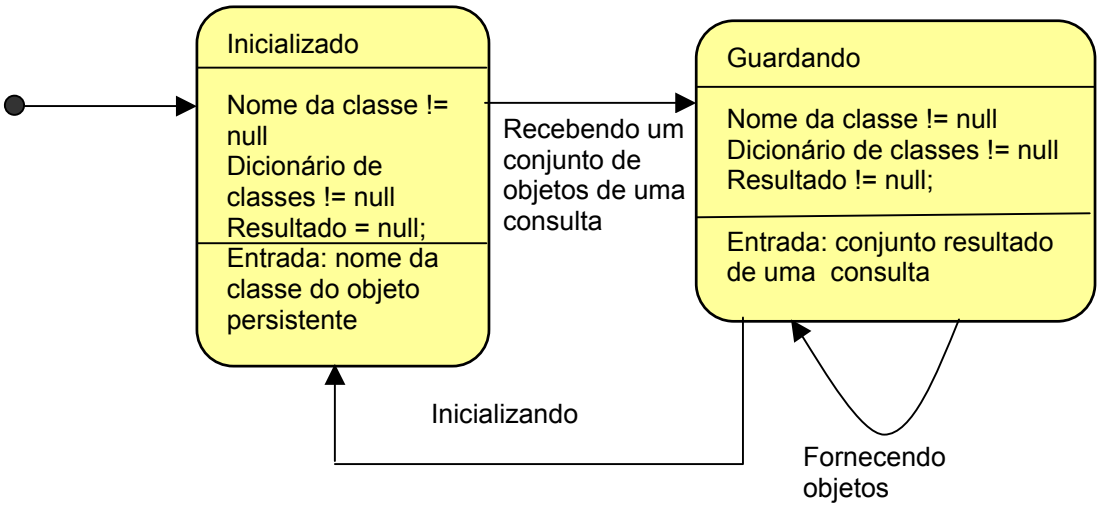
Figura 30: Diagrama de estados para a classe PersistentCriteria



Classe PersistentSet

Os objetos do tipo PersistentSet auxiliarão na armazenagem e gerenciamento de uma coleção de objetos retornados em uma consulta.

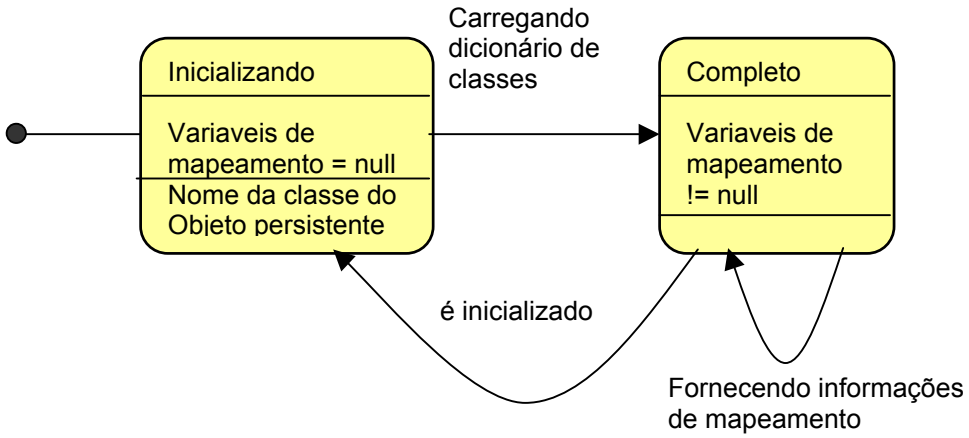
Figura 31: Diagrama de estados para a classe PersistentSet



Classe ClassDictionary

Nesta classe os objetos fornecerão acesso às informações sobre o dicionário de classes e sobre o mapeamento entre atributos e colunas.

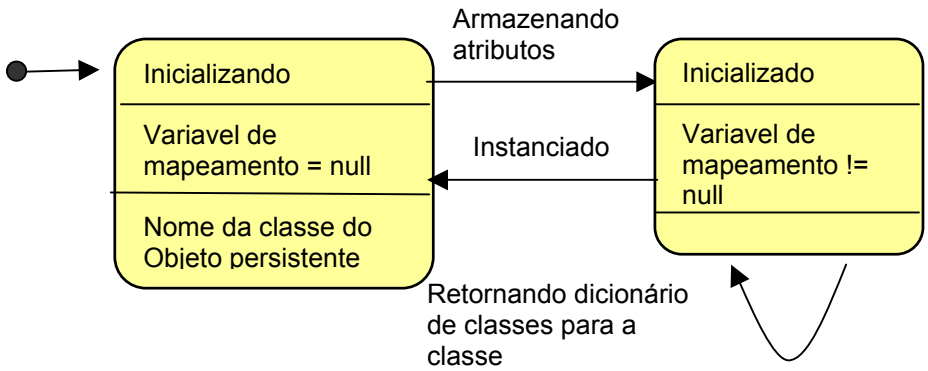
Figura 32: Diagrama de estados para a classe ClassDictionary.



Classe DataDictionary

Os objetos desta classe fornecerão acesso para se buscar o mapeamento entre classes e tabelas.

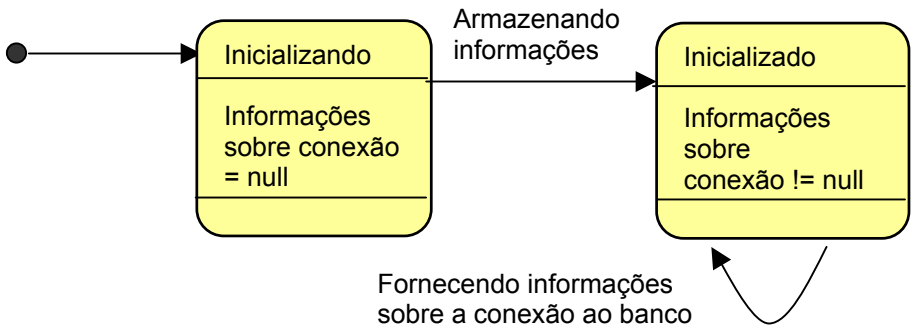
Figura 33: Diagrama de estados para a classe DataDictionary.



Classe Database

Os objetos do tipo Database armazenarão as informações pertinentes a conexão ao banco de dados, fornecendo mecanismos de conexão ao banco para outras classes da camada.

Figura 34: Diagrama de estados para a classe Database.



4.4.4 Diagramas de seqüência

Para descrever o comportamento dinâmico do domínio de classes, serão usados os diagramas de seqüência. As bases para a construção dos diagramas de seqüência são os casos de uso, onde cada caso de uso tem que ser descrito com o seu impacto no domínio de classes, ilustrando como o domínio de classes colabora para executar um caso de uso dentro do sistema.

A seguir são mostrados os diagramas de seqüência para os casos de uso da camada de persistência.

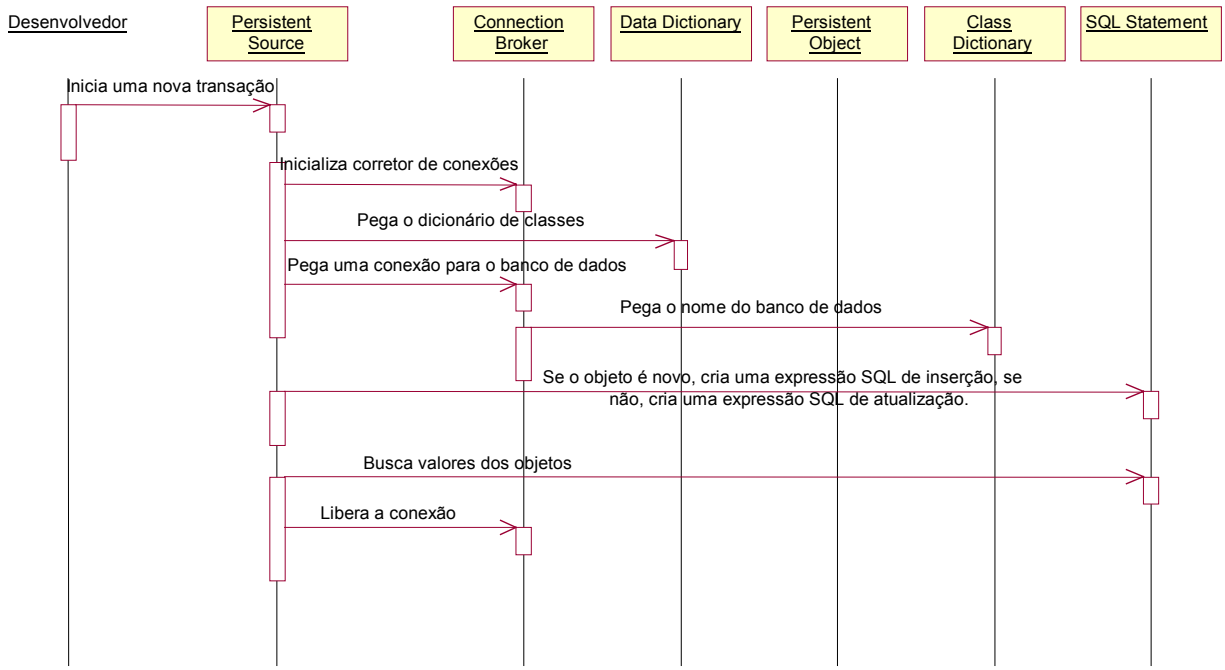
Caso de Uso 1 Cenário 1:

Salvar ou atualizar um objeto para um mecanismo persistente

Salvar ou atualizar um objeto persistente pode ser feito através de uma transação (onde a operação *save* fará apenas parte de um conjunto de operações sobre o banco, que poderão suceder ou falhar como um todo) ou simplesmente através de uma operação isolada, fora de uma transação.

Sem fazer parte de uma transação, o desenvolvedor utilizará o comando *save* tanto para o caso de salvar um objeto, quanto para o caso de se atualizar um objeto no banco de dados. A camada terá a tarefa de saber se o objeto persistente já existe ou não no banco de dados, para então saber, se vai criar uma expressão SQL para inserção ou para atualização.

Figura 35: Diagrama de seqüência caso 1 cenário 1 (fase de análise).

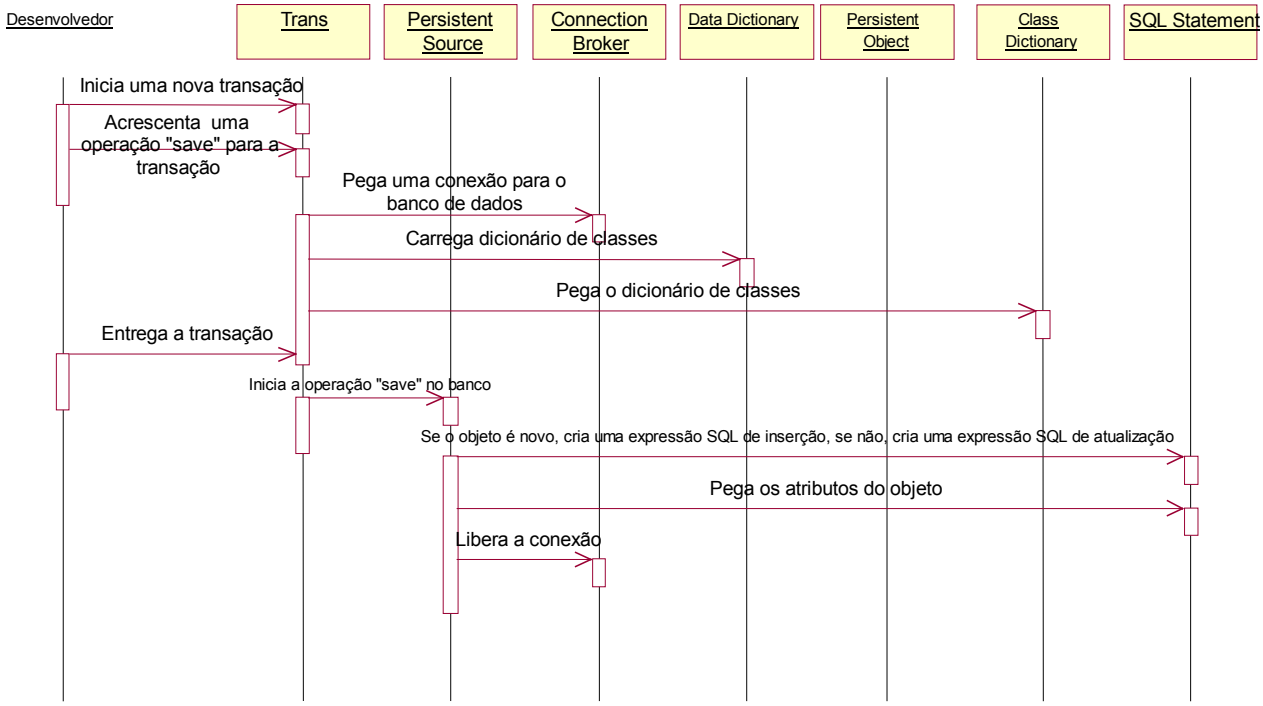


Caso de Uso 1 Cenário 2:

Salvar ou atualizar um objeto para um mecanismo persistente fazendo parte de uma transação

Este caso de uso pode ser executado também através de uma transação. Assim, o comando para salvar ou atualizar um objeto fará parte de um conjunto de operações que podem suceder ou falhar como um todo.

Figura 36: Diagrama de seqüência caso 1 cenário 2 (fase de análise).

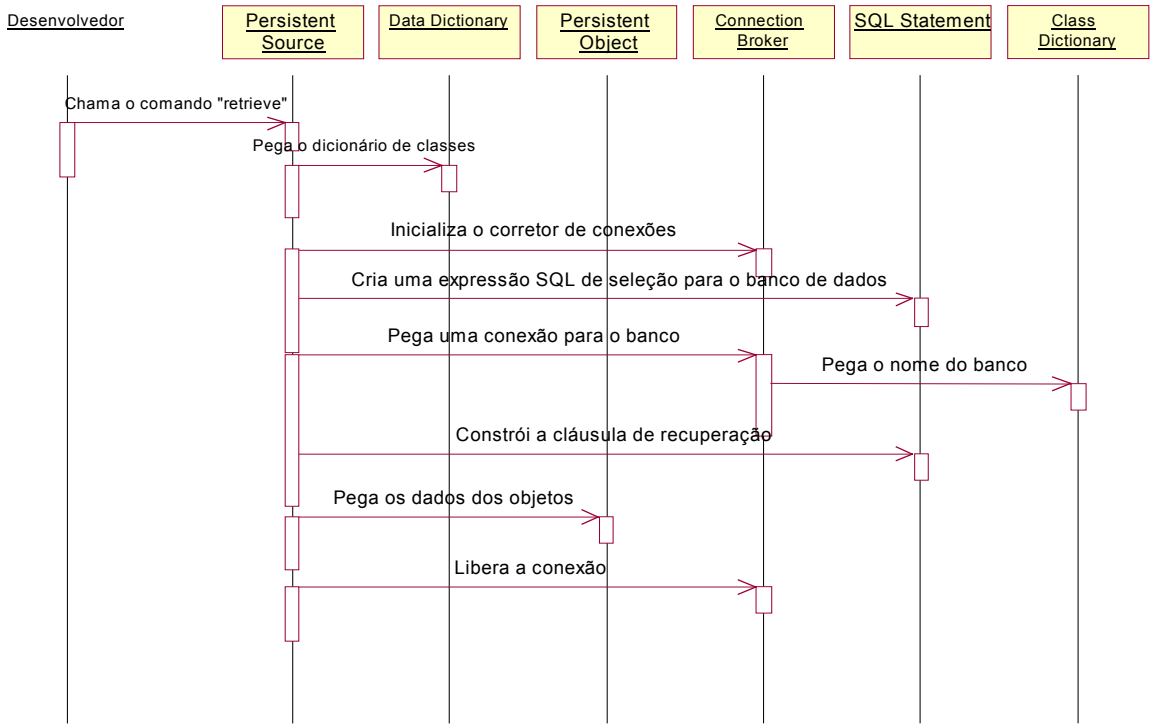


Caso de Uso 2:

Recuperar um objeto de um mecanismo persistente

O desenvolvedor utilizará o comando *retrieve* para o caso de recuperar um objeto. Sem usar um critério de recuperação, usa-se apenas o identificador do objeto para recuperá-lo.

Figura 37: Diagrama de seqüência caso 2 (fase de análise).

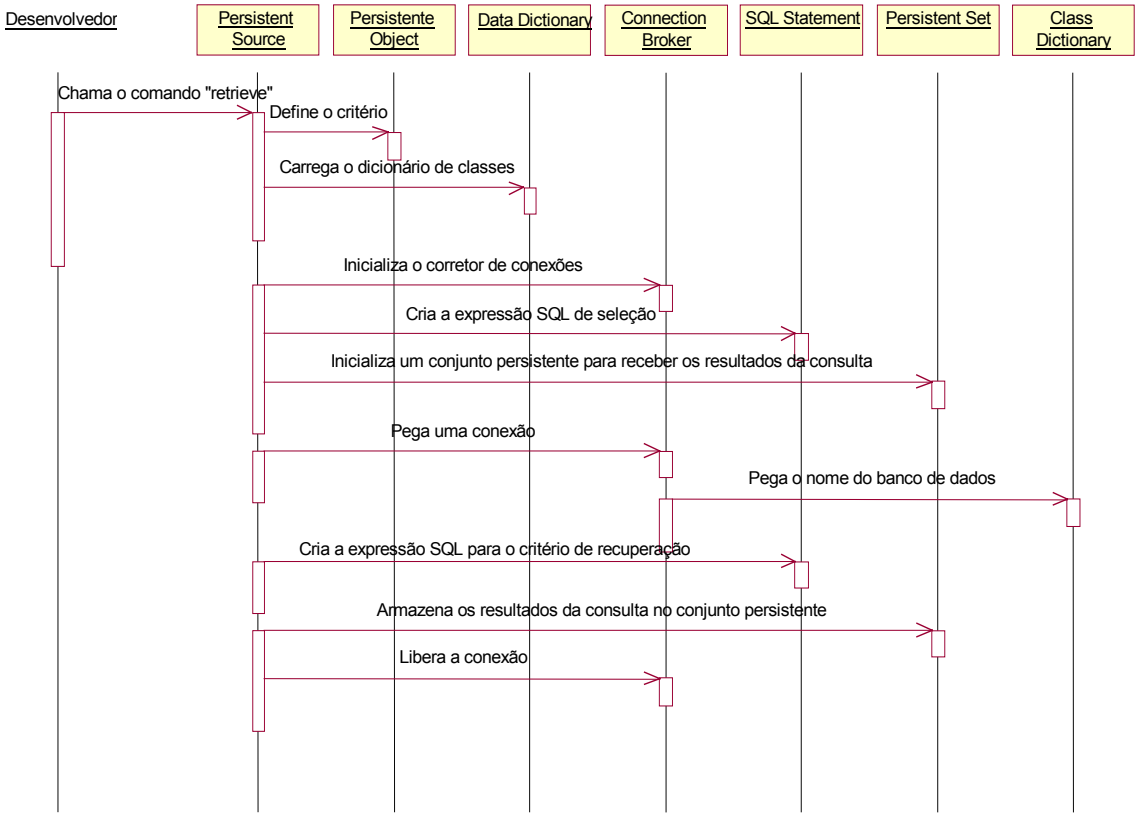


Caso de Uso 3:

Recuperar uma coleção de objetos de um mecanismo persistente

O desenvolvedor utilizará também o comando *retrieve* para o caso de recuperar uma coleção de objetos, mas para isso terá que definir um critério de seleção.

Figura 38: Diagrama de seqüência caso 3 (fase de análise).

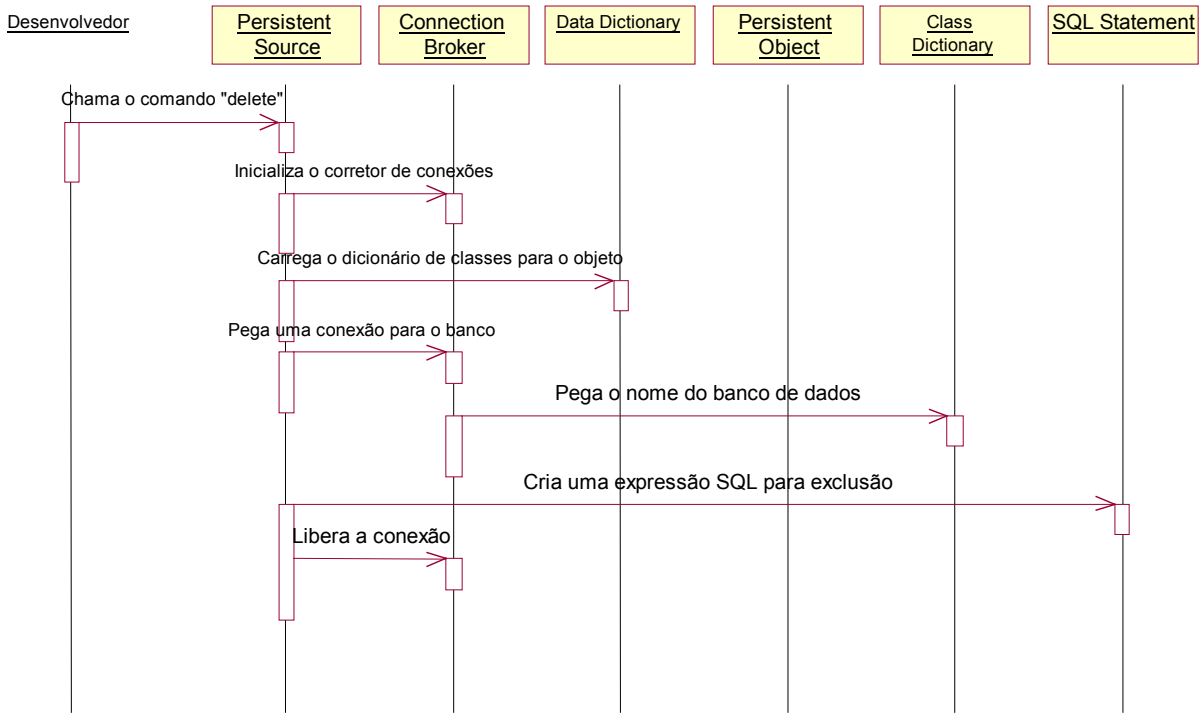


Caso de Uso 4 Cenário 1:

Excluir um objeto de um mecanismo persistente

O desenvolvedor deverá usar o comando *delete* quando quiser isoladamente excluir um objeto do banco de dados. Sem utilizar um critério de exclusão, um objeto é excluído utilizando o seu identificador.

Figura 39: Diagrama de seqüência caso 4 cenário 1 (fase de análise).

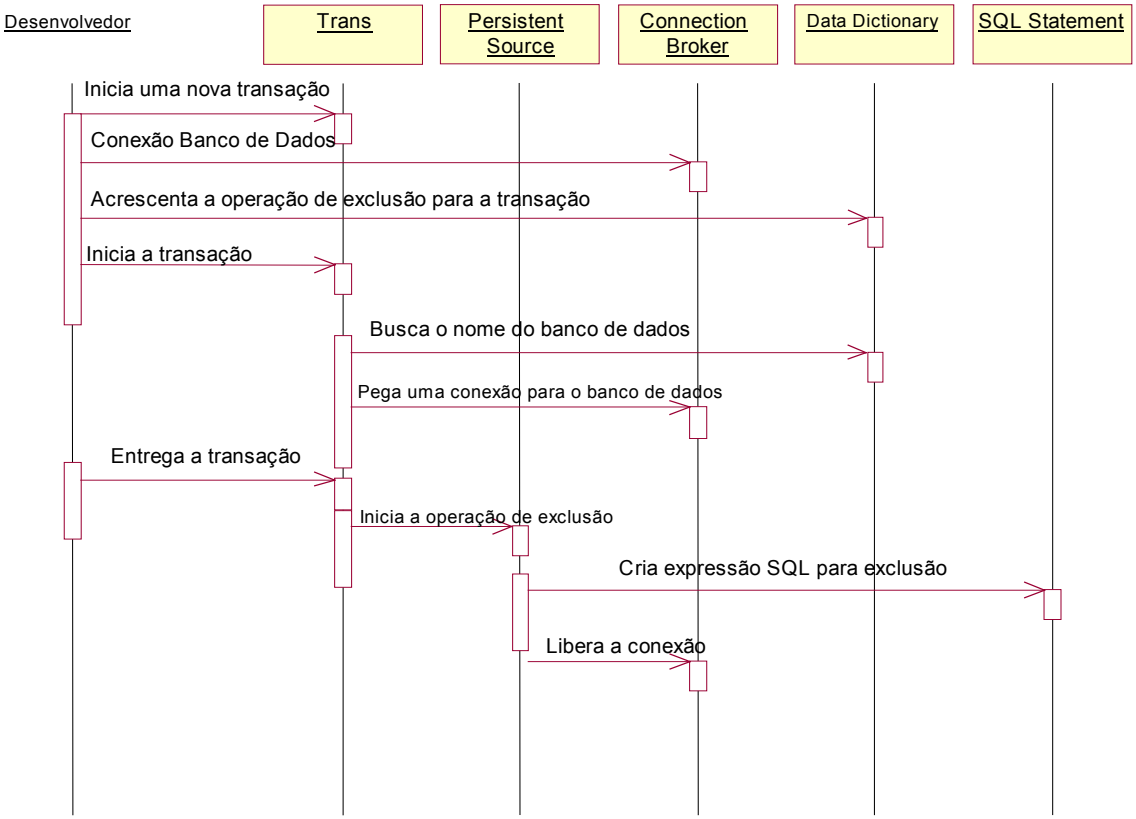


Caso de Uso 4 Cenário 2:

Excluir um objeto de um mecanismo persistente fazendo parte de uma transação

Este pode ser executado também através de uma transação. Assim, o comando para excluir um objeto fará parte de um conjunto de operações que podem suceder ou falhar como um todo.

Figura 40: Diagrama de seqüência caso 4 cenário 2 (fase de análise).

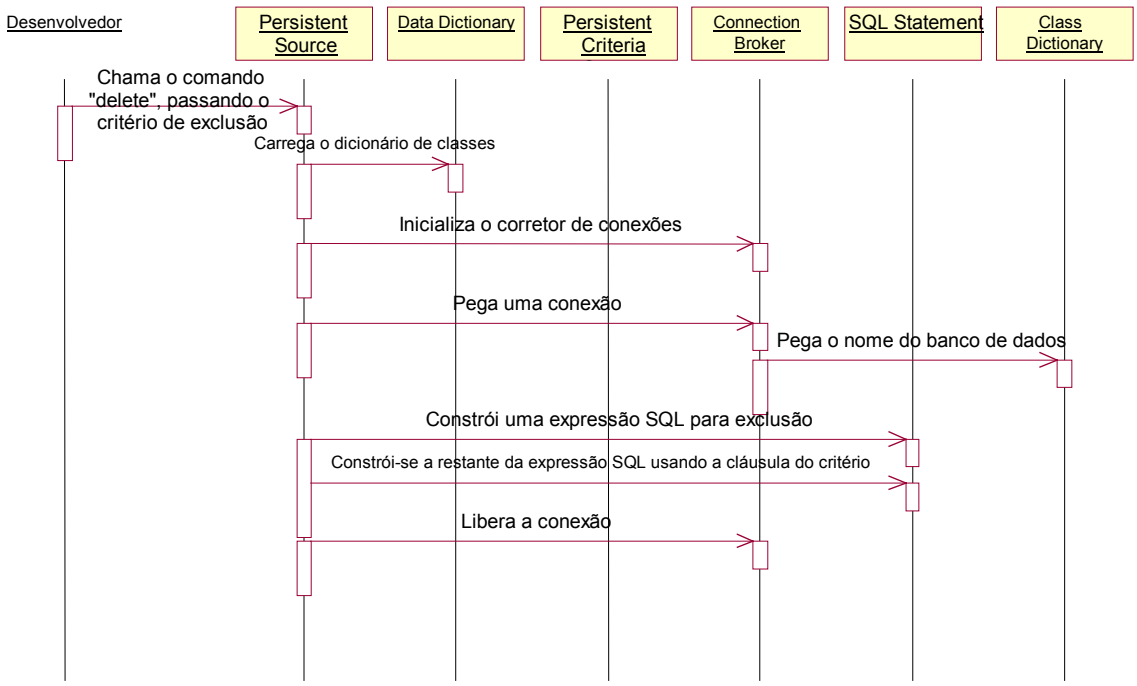


Caso de Uso 5 Cenário 1:

Excluir uma coleção de objetos de um mecanismo persistente

O desenvolvedor utilizará também o comando *delete* para o caso de excluir uma coleção de objetos, mas para isso terá que definir um critério de exclusão.

Figura 41: Diagrama de seqüência caso 5 cenário 1 (fase de análise).

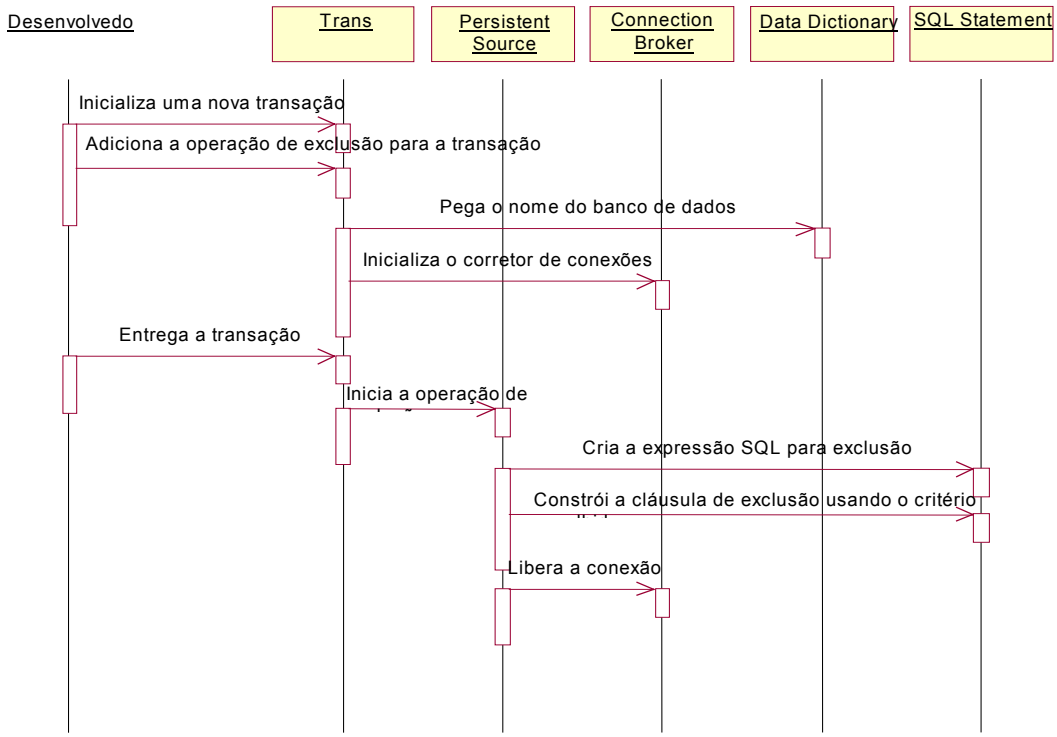


Caso de Uso 5 Cenário 2:

Excluir uma coleção de objetos de um mecanismo persistente fazendo parte de uma transação

Esta operação poderá fazer parte também de uma transação. Para isso, o desenvolvedor utilizará também o comando *delete* para o caso de excluir uma coleção de objetos, mas para isso terá que definir um critério de exclusão.

Figura 42: Diagrama de seqüência caso 5 cenário 2 (fase de análise).



4.5 Projeto

Nesta fase se começa a implementar nos modelos existentes, os melhoramentos e técnicas de como realmente cada função do sistema será concebida. Serão modelos mais detalhados com ênfase nas soluções para armazenamento de dados e funções primordiais do sistema. A fase de projeto pode ser dividida em outras duas fases:

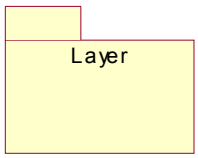
Projeto da arquitetura: Projeto de alto nível onde os pacotes (subsistemas) são definidos, incluindo as dependências e mecanismos de comunicações entre eles. Naturalmente, o objetivo é criar uma arquitetura simples e clara, onde as dependências sejam poucas e que possam ser bidirecionadas, sempre que possível.

Projeto detalhado: Detalha o conteúdo dos pacotes, onde todas classes serão totalmente descritas, com especificações claras para o programador que irá gerar o código da classe. Modelos dinâmicos da UML são usados para demonstrar como os objetos se comportam em diferentes situações.

4.5.1 Projeto da arquitetura

Uma arquitetura bem projetada é a base para futuras expansões e modificações no sistema. Os pacotes podem ser responsáveis por funções lógicas ou técnicas no sistema. É de vital importância separar a lógica da aplicação da lógica técnica. Isso facilitará futuras mudanças no sistema. A definição dos pacotes é mostrada na Figura 43.

Figura 43: Fase de projeto - Definição dos pacotes.

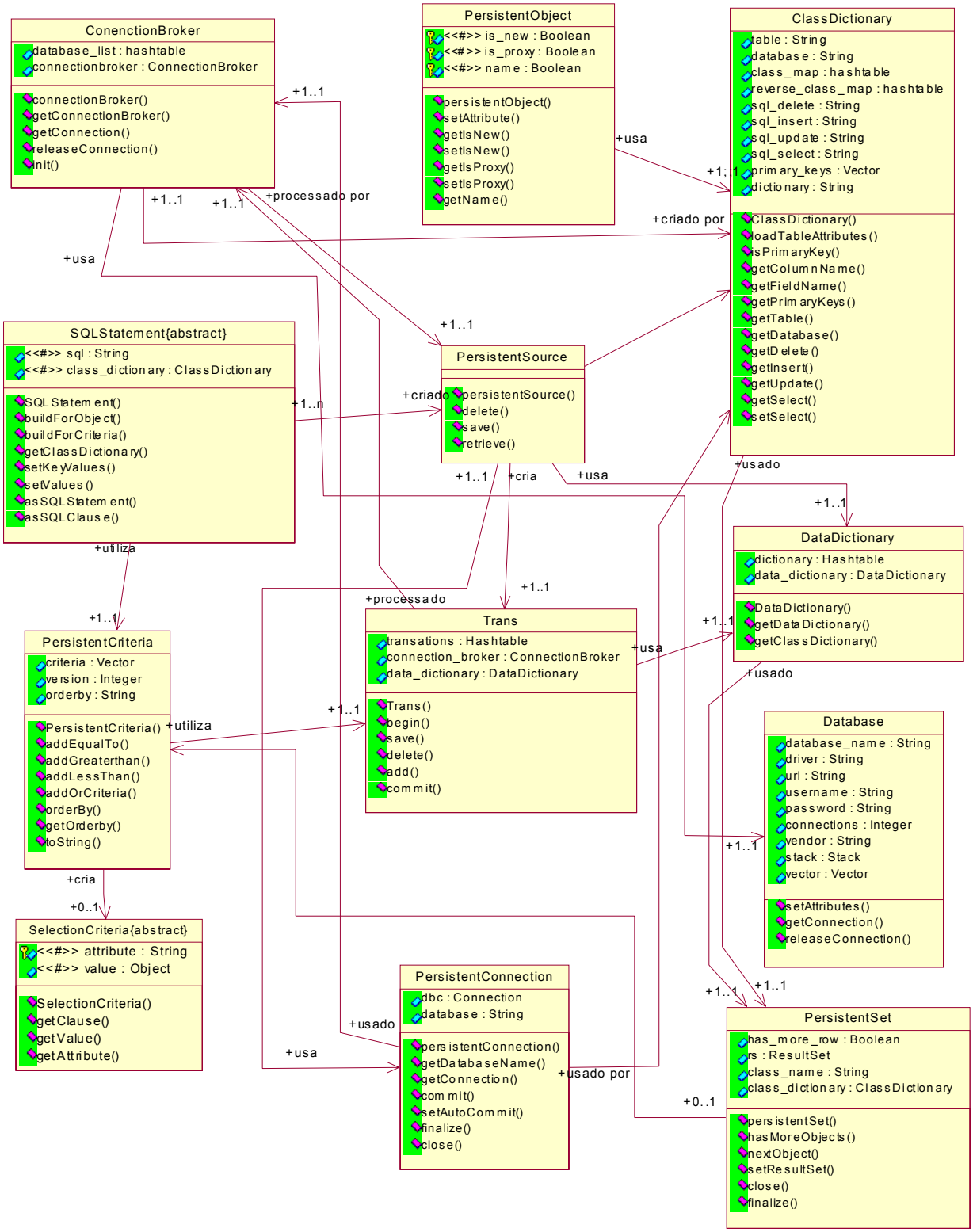


A camada de persistência estará toda contida em um único pacote. Nele estarão contidas as classes no qual o desenvolvedor terá acesso para chamar comandos, tais como *save, delete e retrieve*. Estarão também contidas as classes que manipulam todo o mecanismo responsável por mapear classes / atributos para tabelas / colunas em um banco de dados e vice-versa. Farão parte também do pacote, as classes responsáveis por criar expressões SQL para acesso ao banco de dados com a finalidade de prover as principais operações da camada de persistência.

4.5.2 Projeto detalhado

O propósito do projeto detalhado é descrever as classes com todas as suas funcionalidades. O diagrama de classes terá um detalhamento bem mais técnico e bem mais elevado do que o diagrama de classes da fase de análise. O diagrama de classes é mostrado na figura 44.

Figura 44: Diagrama de classes (fase de projeto).



4.5.2.1 Descrição das classes

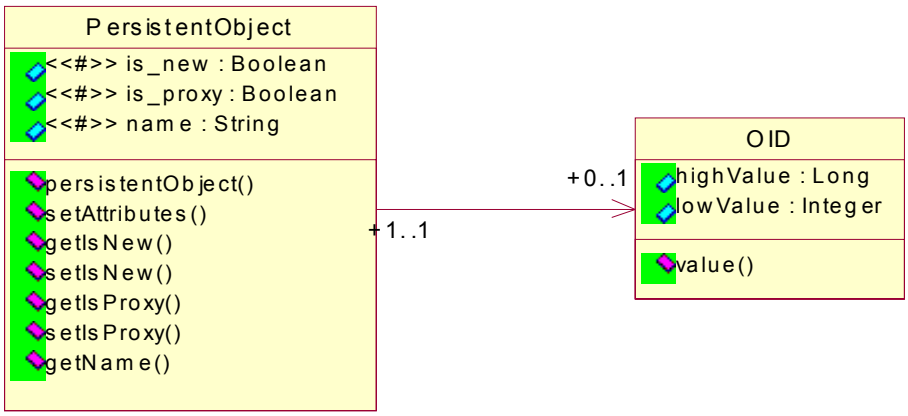
Quadro 11: Descrição das classes

Classe	**Descrição**
ConnectionBroker	Mantém conexões para o mecanismo persistente, no caso deste projeto um banco de dados relacional e gerencia a comunicação entre a aplicação orientada a objetos e o mecanismo persistente.
PersistentConnection	Encapsula o comportamento necessário para criar uma conexão para bancos de dados relacionais.
PersistentObject	Encapsula o comportamento necessário para poder criar instâncias de objetos persistentes, assim, um objeto persistente pode ser salvo, recuperado, atualizado ou excluído de um mecanismo persistente. Ela sabe como armazenar seus atributos de uma linha de resultados de uma tabela do banco de dados.
PersistentSource	É a classe que permite ao desenvolvedor ter acesso para os fundamentais comandos para fazer objetos persistirem .
SQLStatement	Esta hierarquia de classes sabe como construir expressões insert, update, delete e select no padrão SQL(Structured Query Language).
Trans	Permite que um conjunto de operações em objetos sejam agrupadas em uma simples operação que poderá suceder ou falhar como um todo.
SelectionCriteria	Esta hierarquia de classes encapsula o comportamento necessário para criar expressões SQL para recuperar, atualizar ou excluir coleções de objetos baseados em algum critério.
PersistentCriteria	É usada para construir critérios utilizando a hierarquia de classes SelectionCriteria.
PersistentSet	Esta classe encapsula o conceito de cursor de banco de dados.
ClassDictionary	Encapsula o comportamento necessário para mapear classes e atributos em tabelas e colunas em um banco de dados

	relacional.
DataDictionary	Apoia o mapeamento de classes para tabelas em um banco de dados relacional, fornecendo uma completa correspondência entre classes e tabelas.
Database	Encapsula informações sobre um banco de dados e uma conexão utilizando JDBC.

A classe PersistentObject

Figura 45 classe PersistentObject.



A Figura 45 mostra o projeto de duas classes PersistentObject e OID. A classe PersistentObject encapsula o comportamento necessário para poder criar instâncias de objetos persistentes, podendo assim, um objeto persistente ser salvo, recuperado, atualizado ou excluído de um mecanismo persistente. Ela sabe como armazenar seus atributos de uma linha de resultados de uma tabela do banco de dados. Por exemplo, uma classe Cliente indiretamente herdará de PersistentObject. A classe OID encapsula o comportamento necessário para OIDs, usando a aproximação HIGH/LOW para assegurar identificadores únicos.

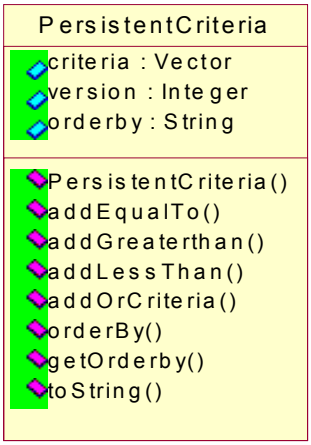
Na classe PersistentObject existem três atributos: is_new, is_proxy e name, no qual indicam respectivamente se um objeto é novo (se ainda não foi ainda salvo no banco de dados relacional), se é um objeto é um proxy e o nome da classe do objeto persistente.

A classe PersistentObject implementa os métodos persistentObject(), setAttributes(), getIsNew(), setIsNew(), getIsProxy(), setIsProxy() e getName(). O método persistentObject inicializa os atributos da classe. O método setAttributes() guarda atributos de objetos a partir de dados recuperados de uma tabela do banco de dados. O método getIsNew() diz se um objeto é novo no banco de dados, já o método setIsNew() marca um objeto como novo. O método getIsProxy() diz se um objeto é proxy, já o método setIsProxy() marca um objeto como proxy. O método getName() tem a função de retornar o nome da classe do objeto persistente.

A classe PersistentObject potencialmente mantém um relacionamento com a classe OID, no qual é feito quando um OID é usado para chaves únicas para objetos no mecanismo persistente.

A classe PersistentCriteria

Figura 46: A classe PersistentCriteria.



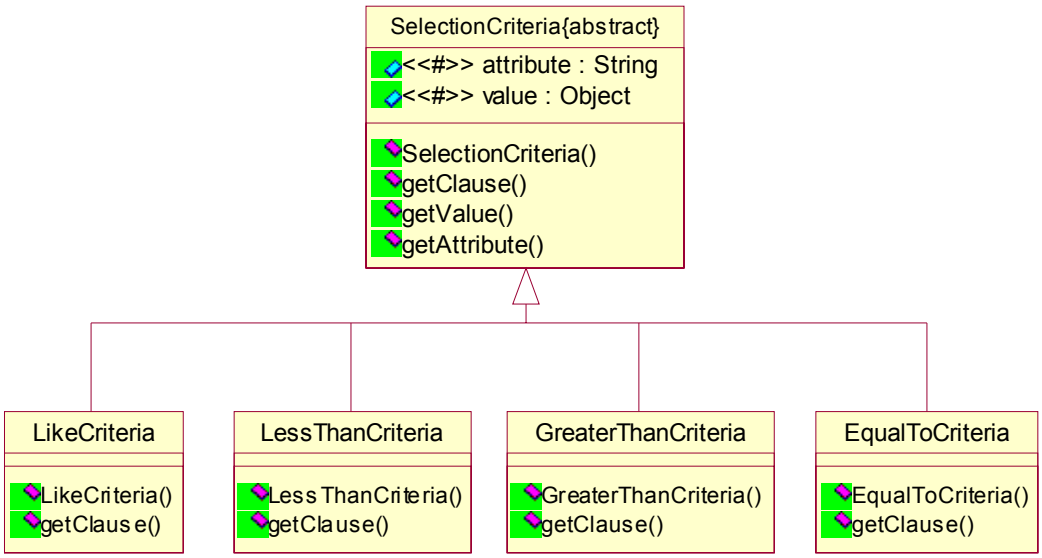
Embora a classe PersistentObject encapsule o comportamento necessário para fazer simples objetos persistirem, ela não é suficiente porque precisa-se também trabalhar com coleções de objetos persistentes. É aí que entra a classe PersistentCriteria, possibilitando recuperar e excluir coleções de objetos de uma só vez, através da definição de um critério de seleção ou exclusão.

A classe PersistentCriteria possui três atributos que são: criteria, version, orderby. O atributo criteria é do tipo Vector e é reponsável por armazenar o critério de seleção usado na montagem da expressão SQL. O atributo version é um inteiro e mostra quantos critérios foram adicionados dentro de uma mesma instância de PersistentCriteria, quer dizer, dentro de uma mesma expressão SQL. O atributo orderby vai receber o nome de uma coluna para ser usada como referência.

A classe PersitentCriteria implementa os métodos: addEqualTo(), addGreaterThan(), addLessThan(), addOrCriteria(), orderBy(), getOrderby(), toString(). O método addEqualTo() adiciona o critério igual(=). O método addGreaterThan() adiciona o critério maior que(Greater Than (>)). O método addLessThan() adiciona o critério menor que(Less Than (<)). O método addOrCriteria junta dois critérios em um só critério através do operador OR. O método orderBy marca uma coluna para ser usada como referência no orderby, já o método getOrderby() dá a coluna que será usada como referência no orderby.

A hierarquia de classes SelectionCriteria

Figura 47: A hierarquia de classes SelectionCriteria.



SelectionCriteria é uma classe abstrata, que captura o comportamento comum de suas subclasses, mas não pode ser instanciada diretamente, a partir dela pode-se limitar o escopo de um pequeno conjunto de objetos. A classe SelectionCriteria é a super classe de todas as classes de critério. A classe SelectionCriteria possui os seguintes atributos: attribute e value. O atributo “attribute” guarda o nome do atributo que será usado como atributo chave no critério de seleção, já o atributo “value” armazena o valor do atributo que será usado como valor chave no critério seleção.

A classe SelectionCriteria implementa os seguintes métodos: getClause(), getValue() e getAttribute(). O método getClause() é abstrato, ele dá a cláusula usada no critério de seleção e é implementado por cada subclasse de SelectionCriteria. O método getValue() dá o valor do atributo que será usado como valor chave no

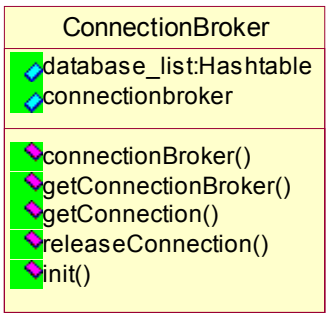
critério de seleção, já o método getAttribute() dá o nome do atributo que será usado como atributo chave no critério de seleção.

A classe LikeCriteria possui além do método construtor, o método getClause() que dá a cláusula “LIKE” a ser utilizada no critério de seleção.

A classe LessThanCriteria possui além do método construtor, o método getClause() que dá a cláusula “<” a ser utilizada no critério de seleção. A classe GreaterThanCriteria possui além do método construtor, o método getClause() que dá a cláusula “>” a ser utilizada no critério de seleção. A classe EqualToCriteria possui além do método construtor, o método getClause() que dá a cláusula “=” a ser utilizada no critério de seleção.

A classe ConnectionBroker

Figura 48: A classe ConnectionBroker.



A classe ConnectionBroker é o ponto chave da camada de persistência, ela é responsável por manter conexões para o mecanismo persistente (banco de dados relacional) durante todo o tempo de execução e gerenciar a comunicação entre a aplicação orientada a objetos e o mecanismo persistente.

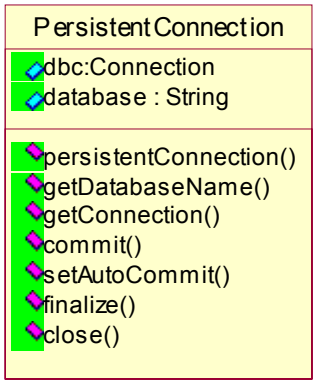
Quando inicia-se uma aplicação, uma das principais tarefas da classe ConnectionBroker é ler a informação necessária para criar instâncias das classes de mapeamento tais como: Database, ClassDictionary e DataDictionary.

A classe ConnectionBroker possui dois atributos: O atributo database_list é usado para armazenar os bancos de dados usados e o atributo connectionbroker que é do tipo ConnectionBroker. A classe ConnectionBroker implementa os métodos: getConnectionBroker(), getConnection(), releaseConnection() e init(). O método getConnectionBroker() retorna uma instância de ConnectionBroker. O método getConnection() dá uma conexão para um específico banco de dados. O

método releaseConnection() libera uma conexão ao banco de dados, libera para o reuso. O método init() inicializa a classe ConnectionBroker.

A classe PersistentConnection

Figura 49: A classe PersistentConnection.

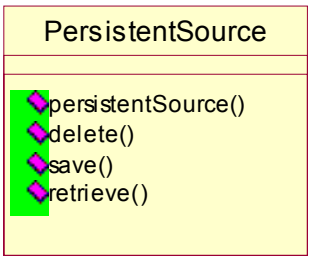


A classe PersistentConnection é responsável por criar uma conexão para bancos de dados relacionais. Esta classe é bem simples e apresenta apenas dois atributos: O atributo dbc que é do tipo Connection e vai armazenar a conexão criada e o atributo database que é uma string recebendo o nome do banco de dados quando ele for informado.

A classe PersistentConnection é formada pelos seguintes métodos: persistentConnection(), getDatabaseName(), getConnection(), commit(), setAutoCommit(), finalize(), close(). O método persistentConnection() é o método Construtor e pega informações tais como driver, url para criar uma conexão JDBC para um banco de dados. O método getDatabaseName() fornece o nome do banco de dados usado na conexão. O método getConnection() retorna uma instância da conexão usada para enviar puro SQL para o banco de dados. O método commit() confirma a transação, já o método SetAutoCommit() seta uma confirmação automática. O método finalize() chama o método close() que fecha a conexão.

A classe PersistentSource

Figura 50: A classe PersistentSource.

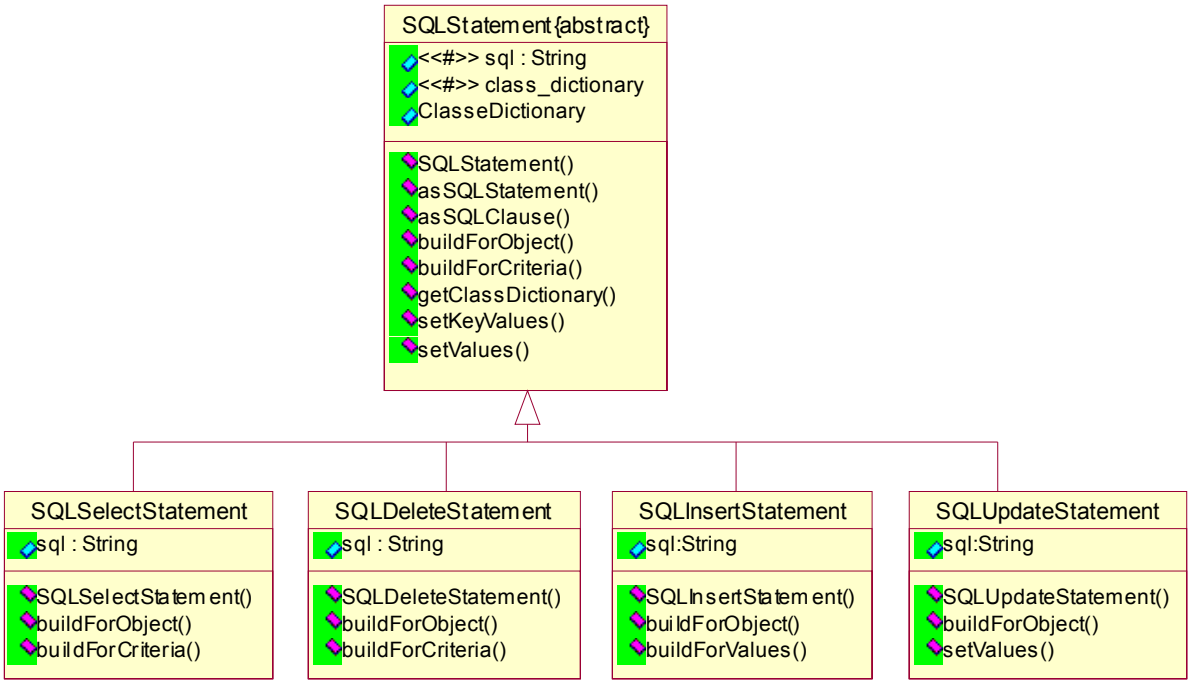


A classe PersistentSource é a principal classe de acesso a camada pelo usuário, é a classe fonte dos métodos que acrescentam os fundamentais comandos para fazer objetos persistirem. Ela fornece comandos para salvar, excluir um objeto ou coleções de objetos (utilizando ou não algum critério de exclusão) e recuperar um objeto ou coleções de objetos (utilizando ou não algum critério de recuperação).

A classe PersistentSource fornece os seguintes métodos: delete(), save() e retrieve(). O método delete() tem a função de excluir objetos de um banco de dados. O método save() tem a função de salvar objetos para um banco de dados. O método retrieve() tem a função de recuperar objetos de um banco de dados.

A hierarquia de classes SQLStatement

Figura 51: A hierarquia de classes SQLStatement.

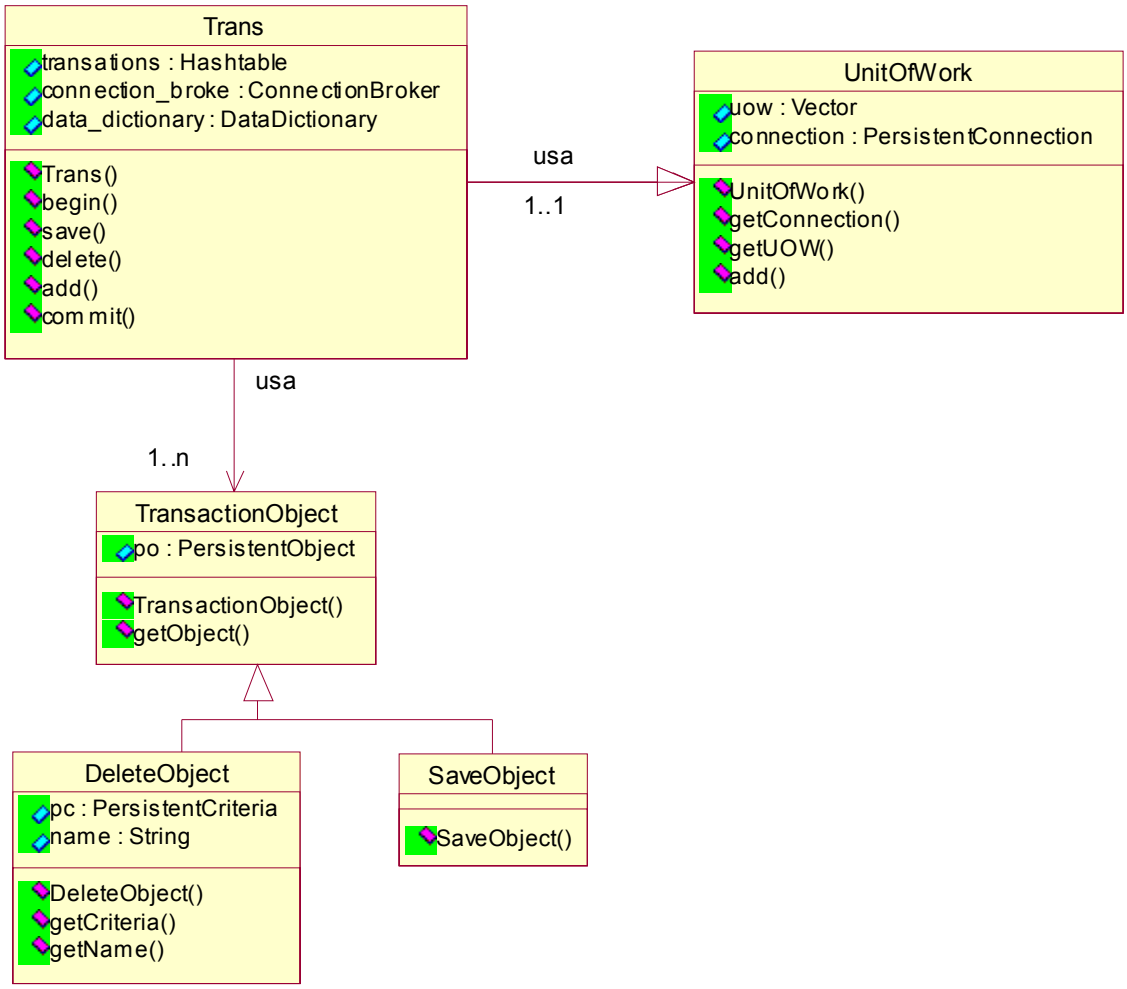


A hierarquia de classes SQLStatement encapsula a habilidade necessária para criar expressões SQL de *SELECT, INSERT, UPDATE e DELETE*. A classe SQLStatement é uma classe abstrata, ela é a super classe das classes SQLSelectStatement, SQLDeleteStatement, SQLInsertStatement e SQLUpdateStatement. A classe SQLStatement possui dois atributos: sql e class_dictionary. O atributo sql é do tipo *string* e vai armazenar a expressão SQL construída, já o atributo class_dictionary é do tipo ClassDictionary e vai servir para buscar a coluna de uma determina tabela, que deve ser usada na criação de alguma expressão SQL. A classe SQLStatement implementa os seguintes métodos: SQLStatement(), asSQLStatement(), asSQLClause(), buildForObject(), buildForCriteria(), getClassDictionary(), setKeyValues(), setValues(). O método SQLStatement() é o construtor da classe. O método getClassDictionary() retorna o dicionário de classes usado pela expressão SQL. O método buildForObject() é abstrato e toda subclasse deve implementar o método buildForObject(), este método constrói uma expressão SQL para um objeto como uma *string*. O método buildForCriteria() constrói uma expressão SQL para um objeto restringida por algum critério. O método asSqlClause() muda uma expressão para um StringBuffer, por questões de eficiência. O método setKeyValues() guarda valores de chaves primárias em uma expressão preparada (prepared statement) a partir de valores dos atributos do objeto persistente. O método setValues() é chamado somente antes de uma execução JDBC para inserir os correntes valores de um objeto.

As classes filhas SQLSelectStatement, SQLDeleteStatement, SQLInsertStatement e SQLUpdateStatement sobrecarregam métodos como buildForCriteria, buildForObject e setValues para atenderem às suas necessidades particulares.

A classe Trans

Figura 52: A Classe Trans



O projeto da figura 52 mostra cinco classes: a Trans, a TransactionObject, a SaveObject, a DeleteObject e a UnitOfWork.

A classe Trans permite que um conjunto de operações em objetos sejam agrupadas em uma simples operação que pode suceder ou falhar como um todo. Ela é a classe principal deste diagrama de classes. A classe Trans possui três atributos: transactions, connection_broker e data_dictionary. O atributo transactions armazena todas as operações sobre o banco que fazem parte de uma transação. Os atributos connection_broker e data_dictionary servem respectivamente para instanciar as classes ConnectionBroker e DataDictionary.

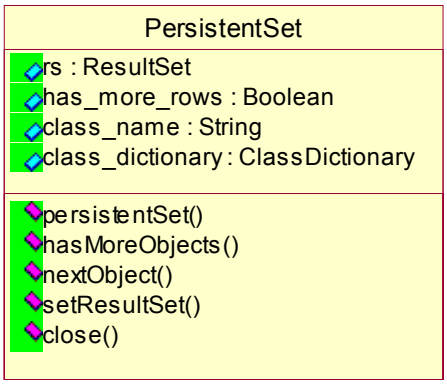
A classe Trans implementa os seguintes métodos: Trans(), begin(), save(), delete(), add() e commit(). O método Trans() é o método construtor e tem a função de instanciar connection_broker e data_dictionary. O método begin() inicia uma nova

transação, esquecendo uma velha transação. O método save() acrescenta um objeto para ser salvo usando uma unidade de trabalho, já o método delete() acrescenta um objeto para ser excluído usando uma unidade de trabalho. O método add() acrescenta um objeto para unidade de trabalho (UnitOfWork). O método commit() é responsável pela entrega da transação.

A classe UnitOfWork tem a função de gerenciar transações para cada banco de dados utilizado. A classe TransactionObject é a super classe das classes SaveObject e DeleteObject. As classes SaveObject e DeleteObject têm a função de especificar qual será a operação executada dentro de uma unidade de trabalho, se é salvar ou excluir.

A classe PersistentSet

Figura 53: A classe PersistentSet.



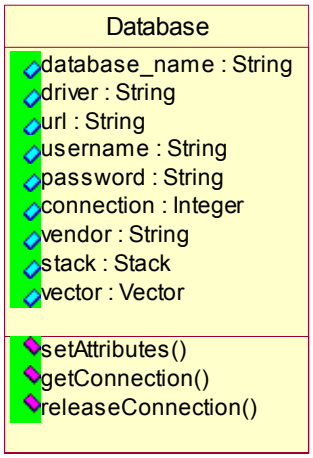
A classe PersistentSet encapsula o conceito de cursor de banco de dados. Cursores permitem a recuperação de subconjuntos de dados de um banco de dados de uma só vez. Esta classe possui quatro atributos principais que são: rs, has_more_rows, class_name e class_dictionary. A atributo rs é do tipo ResultSet e armazena o conjunto resultado de uma consulta ao banco de dados. O atributo has_more_rows diz se ainda existe alguma linha resultante de uma consulta ao banco de dados. O atributo class_name representa o nome da classe do subconjunto de objetos. O atributo class_dictionary é do tipo ClassDictionary e serve para instanciar a classe ClassDictionary para obter o nome da tabela para uma dada classe.

A classe PersistentSet possui os seguintes métodos: hasMoreObjects(), nextObject(), setResultSet(), close() e finalize(). O método hasMoreObjects() é usado para retornar a variável que diz se tem mais linhas resultantes de uma consulta ao banco de dados. O método nextObject() retorna o próximo objeto no

conjunto de objetos retornados depois de uma consulta ao banco de dados. O método setResultSet() serve para marcar o valor do atributo has_more_rows. O método close() libera os recursos do banco de dados no fim da consulta, já o método finalize() é usado para finalizar a operação.

A classe Database

Figura 54: A classe Database.

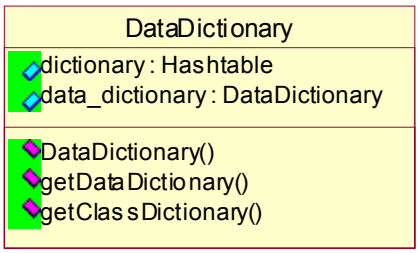


A classe Database encapsula o comportamento necessário para mapear objetos para mecanismos persistentes. A classe Database encapsula informações sobre um banco de dados e uma conexão utilizando JDBC. A classe Database possui os seguintes atributos que servem para identificar o banco de dados e a conexão: database_name, driver, url, username, password, connections, vendor. Existem mais dois atributos que são: stack e vector que são responsáveis por alocar as conexões para o banco de dados.

A classe Database possui quatro métodos que são: O método setAttributes() que guarda as informações retornadas da consulta ao banco (tabela Database) nos atributos e cria uma conexão para com estes dados e armazena na pilha(stack). O método getConnection() que retorna uma conexão para o banco de dados. O método releaseConnection() que libera uma conexão ao banco de dados.

A classe DataDictionary

Figura 55: A classe DataDictionary.

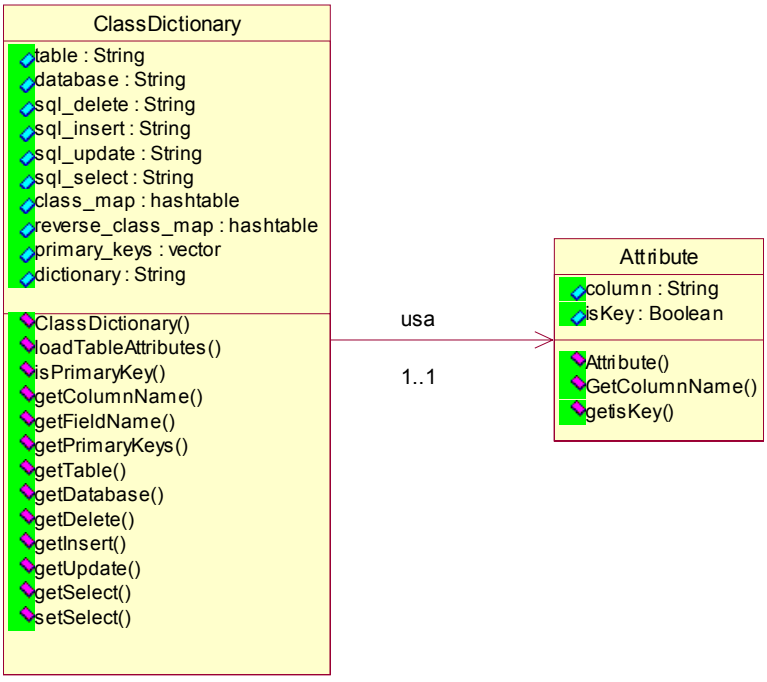


A classe DataDictionary é uma das classes de mapeamento, que encapsulam o comportamento necessário para mapear objetos para mecanismo persistente. Esta classe faz a correspondência entre classes e tabelas em um banco de dados. A classe DataDictionary possui dois atributos: O atributo data_dictionary vai armazenar uma instância da classe DataDictionary. O atributo dictionary armazenará uma correspondência entre nome da classe e tabela no banco de dados.

A classe DataDictionary tem três métodos que são: DataDictionary que é o método construtor da classe. O método getDataDictionary retorna uma instância de DataDictionary. O método GetClassDictionary retorna uma referência para a ClasseDictionary para um dado nome de classe.

A classe ClassDictionary

Figura 56: A classe ClassDictionary



.

A classe ClassDictionary é também uma das classes de mapeamento, que encapsulam o comportamento necessário para mapear objetos para mecanismo persistente. Esta classe faz a correspondência entre atributos e colunas em um banco de dados. A classe ClassDictionary possui os seguintes atributos: table, database, sql_delete, sql_insert, sql_update, sql_select, class_map, reverse_class_map, primary_keys, dictionary. Os atributos table e database vão armazenar respectivamente o nome de uma tabela e o nome de um banco de dados após uma consulta a tabela de dicionário de dados(datadictionary table). Os atributos sql_delete, sql_insert, sql_update, sql_select são strings que vão quardar respectivamente uma expressão SQL para uma básica exclusão, inserção, atualização e seleção. Os atributos class_map, reverse_class_map armazenam respectivamente o mapeamento entre atributos-colunas e colunas-atributos. O atributo primary_keys é um vector e tem a função de armazenar as chaves primárias de cada tabela no banco de dados. O atributo dictionary vai ter a função de armazenar o nome da classe do objeto persistente.

A classe ClassDictionary implementa os seguintes métodos: O método ClassDictionary() que é o método construtor e carrega um dicionário de dados (Data_Dictionary) para uma classe específica. O método loadTableAttributes() que carrega os atributos para uma classe em particular. O método isPrimaryKey() que retorna true se um atributo é uma chave primária de uma tabela. O método getColumnName() que retorna o nome da coluna de uma tabela no banco de dados. O método String getFieldName() retorna o nome do campo de um objeto persistente para um dado nome de coluna de uma tabela no banco de dados. O método getPrimaryKeys() retorna uma enumeração de nomes de chaves primárias da class_dictionary. O método getTable() retorna o nome da tabela no banco de dados correspondente a objeto persistente. O método getDatabase() retorna um nome de um banco de dados onde o objeto é persistido. O método getClassName() retorna o nome da classe do objeto persistente através do atributo dictionary. Os métodos getDelete(), getInsert(), getUpdate() e getSelect() retornam respectivamente uma string que é expressão SQL básica para uma exclusão, inserção, atualização e seleção. Os métodos setDelete(), setInsert(), setUpdate() e setSelect() guardam respectivamente uma string que é expressão SQL básica para uma básica exclusão, inserção, atualização e seleção. Os métodos getDelete(), getInsert(), getUpdate(), getSelect(), setDelete(), setInsert(), setUpdate() e setSelect() auxiliarão a Classe

PersistentSource que é a principal classe de acesso para o desenvolvedor de aplicações orientadas a objetos.

A classe Attribute tem a função de representar um atributo, pois um atributo tem um nome e pode ser uma chave primária. A classe Attribute possui dois principais atributos que são: column que é uma String e representa uma coluna de uma tabela e o atributo isKey que é do tipo boolean e diz se uma certa coluna é chave primária. A classe implementa os seguintes métodos: O método Attribute() que é o método construtor que armazena o nome da coluna e se ela é chave estrangeira. Os métodos getColumnName() e getIsKey() retornam respectivamente o nome da coluna e se uma coluna é chave primária.

4.5.2.2 Diagramas de seqüência

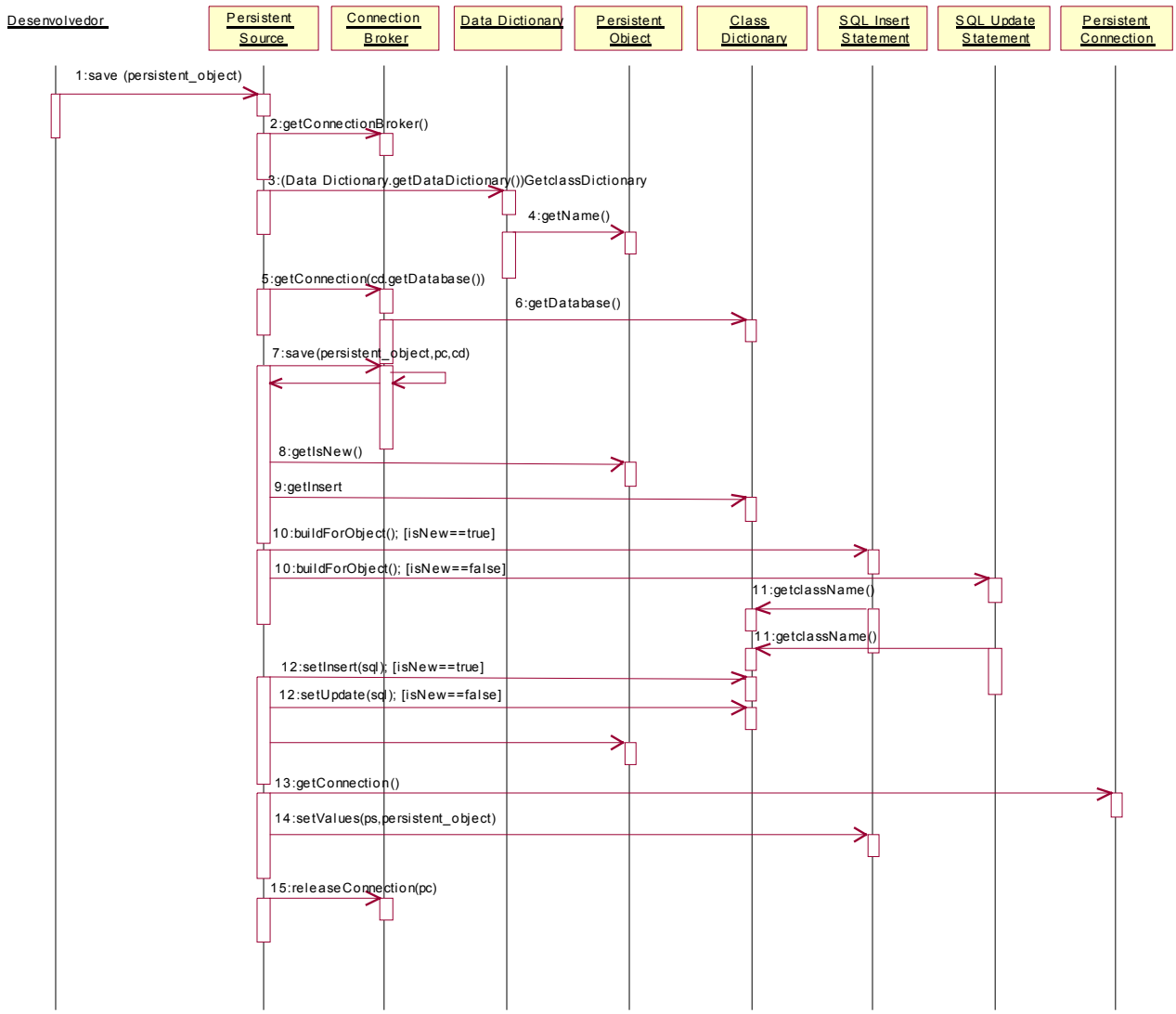
Os diagramas de seqüência de projeto para os casos de uso existentes:

Caso de uso 1: Cenário 1:

Salvar ou atualizar um objeto para um mecanismo persistente.

Aqui a camada terá a tarefa de saber se o objeto persistente já existe ou não no banco de dados, para saber então se vai criar uma expressão SQL para inserção ou para atualização.

Figura 57: Diagrama de seqüência Caso 1 Cenário 1 (fase de projeto).

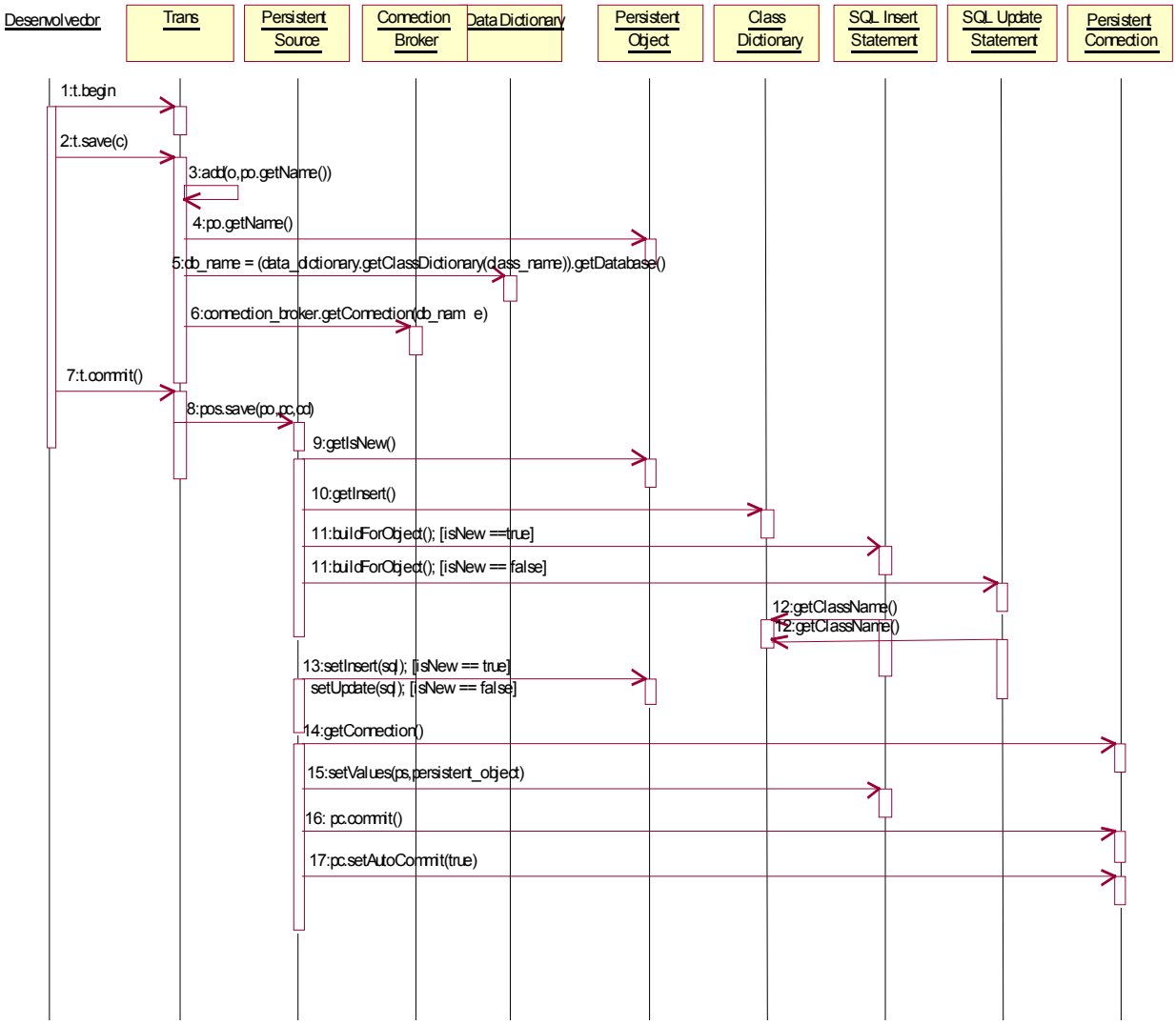


Caso de uso 1: Cenário 2:

Salvar ou atualizar um objeto para um mecanismo persistente fazendo parte de uma transação.

O caso de uso de salvar ou atualizar um objeto pode ser executado também através de uma transação, assim o comando para salvar ou atualizar um objeto fará parte de um conjunto de operações que podem suceder ou falhar como um todo.

Figura 58: Diagrama de seqüência Caso 1 Cenário 2 (fase de projeto).

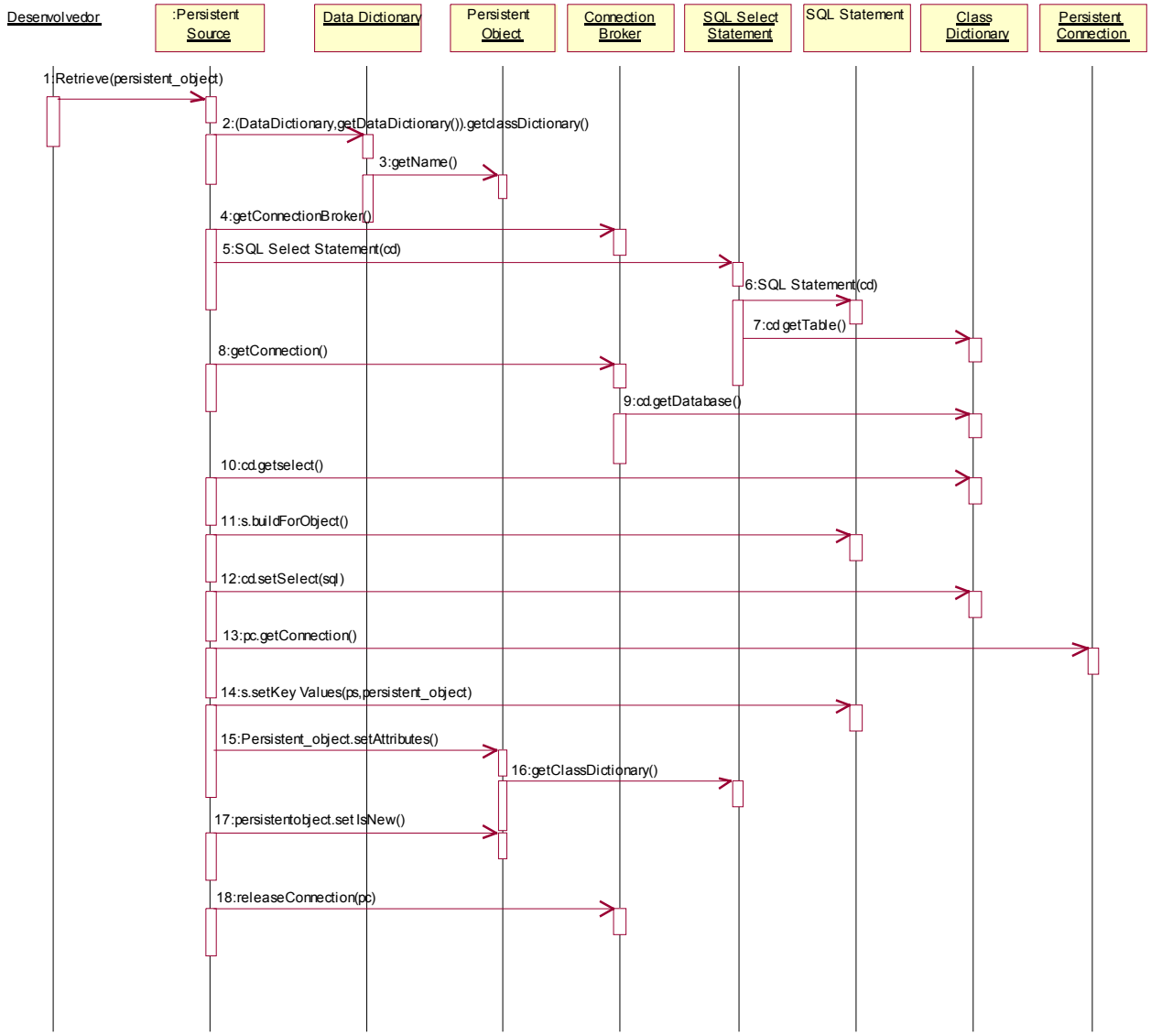


Caso de uso 2:

Recuperar um objeto de um mecanismo persistente.

O desenvolvedor utilizará sempre o comando *retrieve* para o caso de recuperar um objeto. Sem usar um critério de recuperação, usa-se apenas o identificador do objeto para recuperá-lo.

Figura 59: Diagrama de seqüência Caso 2 (fase de projeto).

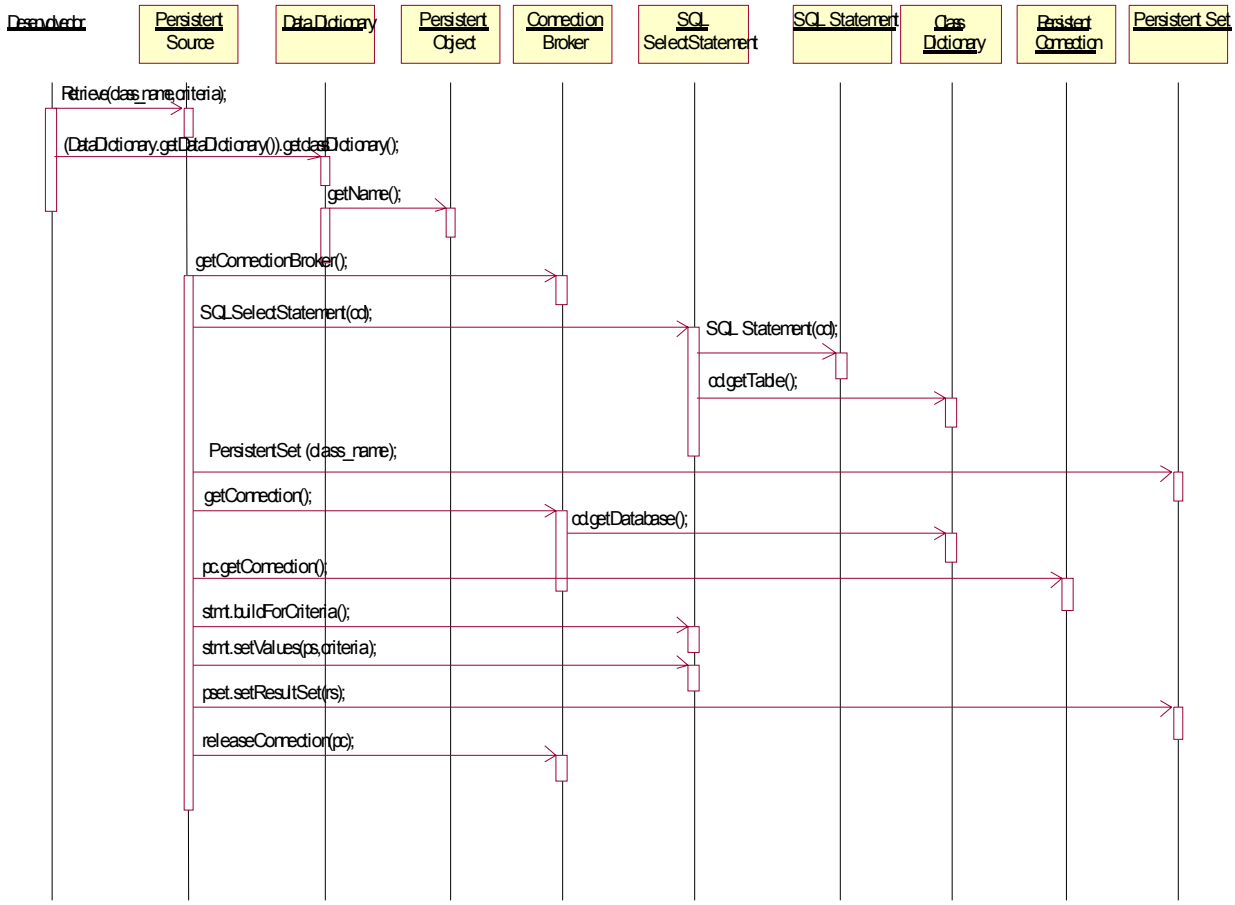


Caso de uso 3:

Recuperar uma coleção de objetos de um mecanismo persistente.

O desenvolvedor utilizará também o comando *retrieve* para o caso de recuperar uma coleção de objetos, mas para isso terá que definir um critério de seleção.

Figura 60: Diagrama de seqüência Caso 3 (fase de projeto).

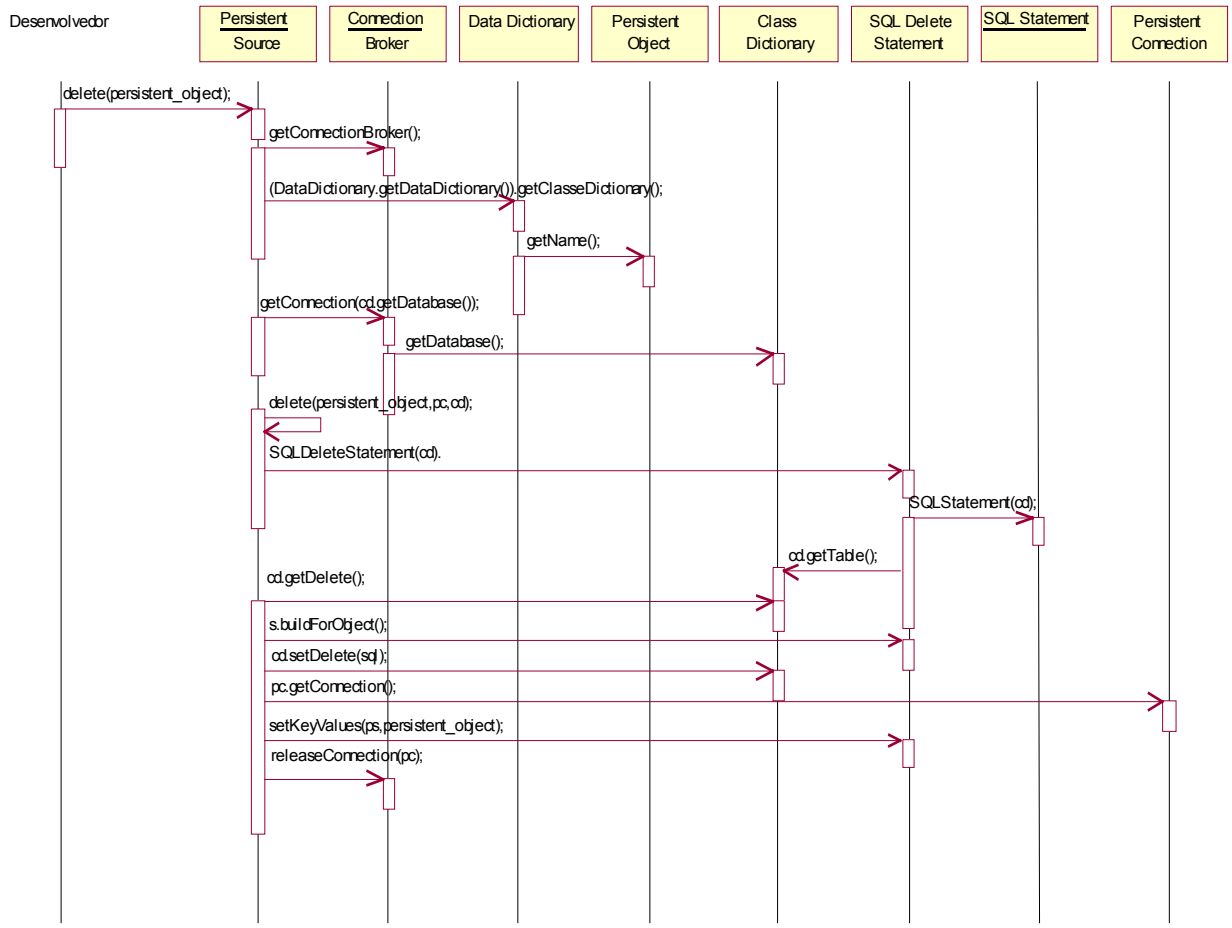


Caso de uso 4: Cenário 1:

Excluir um objeto de um mecanismo persistente.

O desenvolvedor deverá sempre usar o comando *delete* quando quer isoladamente excluir um objeto do banco de dados. Sem utilizar um critério de exclusão, um objeto é excluído utilizando o seu identificador.

Figura 61: Diagrama de seqüência Caso 4 Cenário 1 (fase de projeto).

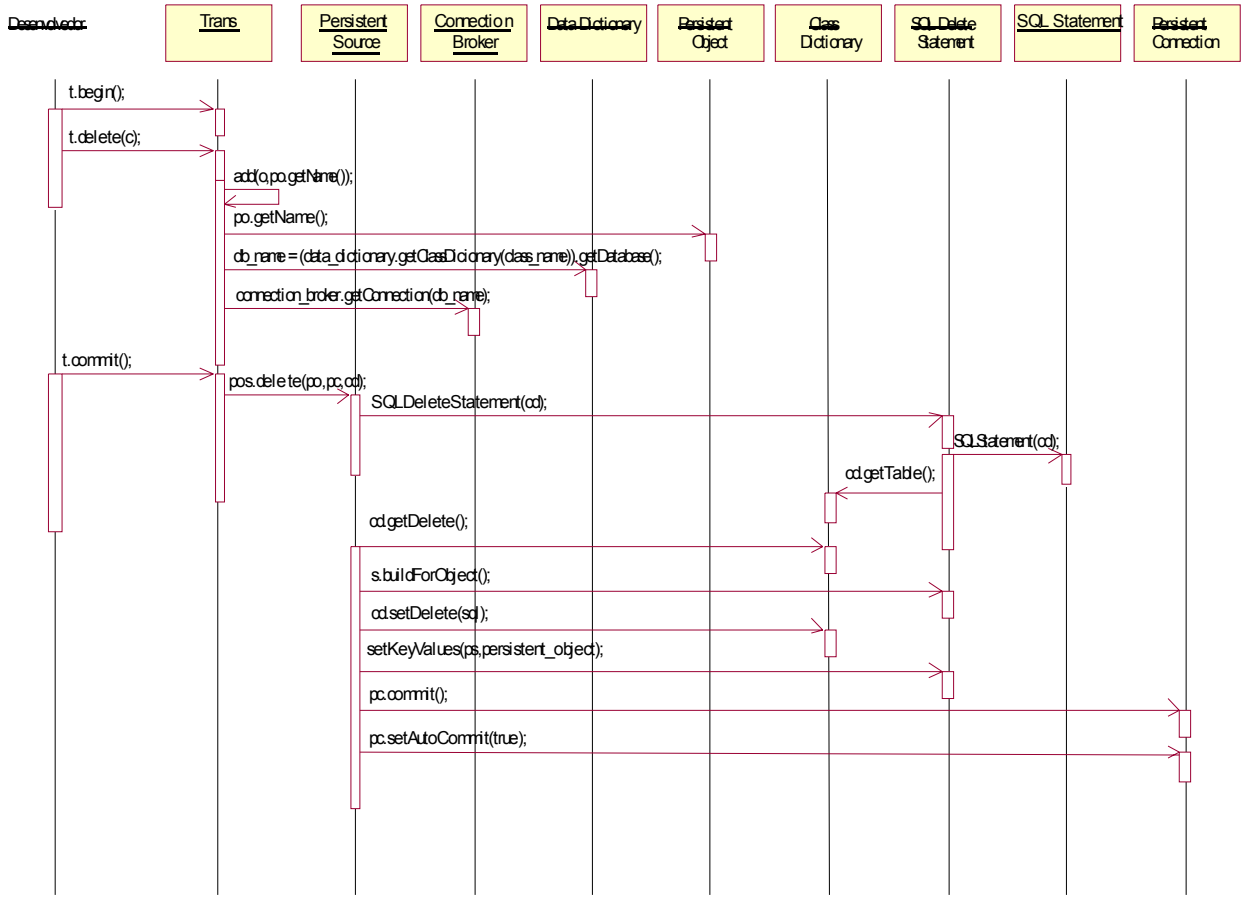


Caso de uso 4: Cenário 2:

Excluir um objeto de um mecanismo persistente fazendo parte de uma transação.

O caso de uso de excluir um objeto pode ser executado também através de uma transação, assim o comando para excluir um objeto fará parte de um conjunto de operações que pode suceder ou falhar como um todo.

Figura 62: Diagrama de seqüência Caso 4 Cenário 2 (fase de projeto).

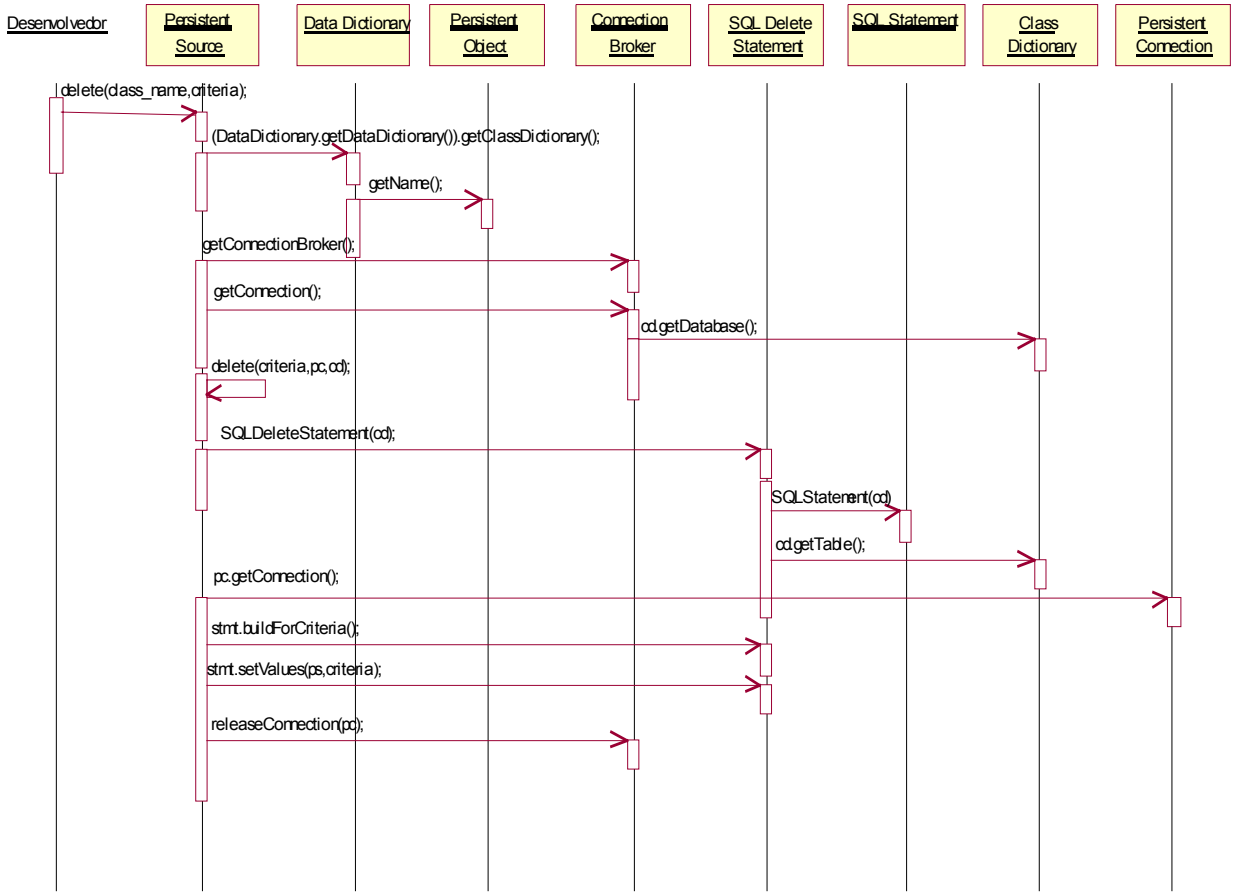


Caso de uso 5: Cenário 1:

Excluir uma coleção de objetos de um mecanismo persistente.

O desenvolvedor utilizará também o comando *delete* para o caso de excluir uma coleção de objetos, mas para isso terá que definir um critério de exclusão.

Figura 63: Diagrama de seqüência Caso 5 Cenário 1 (fase de projeto).

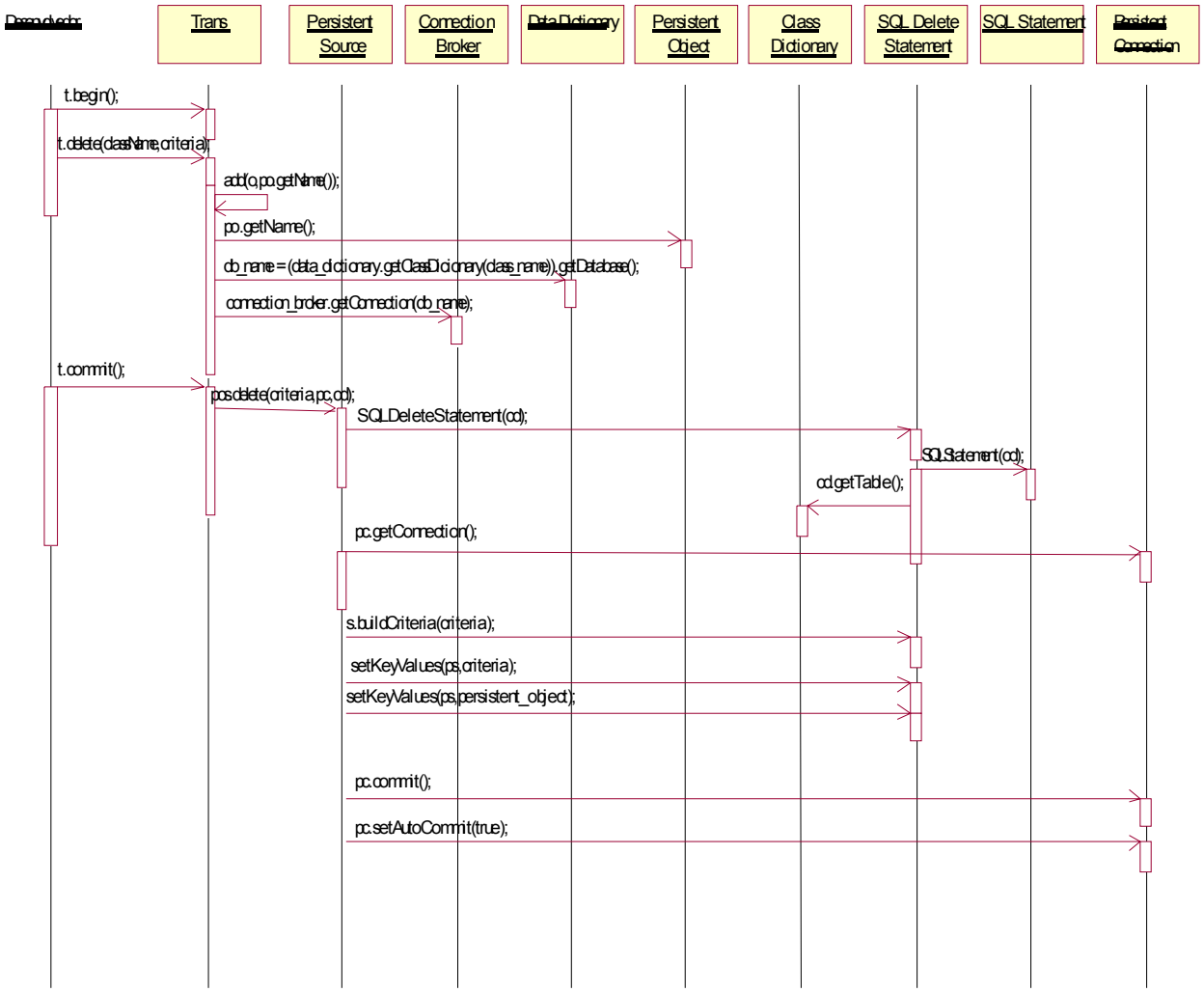


Caso de uso 5: Cenário 2:

Excluir uma coleção de objetos de um mecanismo persistente fazendo parte de uma transação.

Esta operação poderá fazer parte também de uma transação, para isso o desenvolvedor utilizará também o comando delete para o caso de excluir uma coleção de objetos e definirá um critério de exclusão.

Figura 64: Diagrama de seqüência Caso 5 Cenário 2 (fase de projeto).



4.6 Implementação

A implementação da camada de persistência foi feita utilizando a ferramenta JBuilder da Inprise, que constrói aplicações em Java. A implementação foi feita por alunos do curso de graduação em Tecnologia de Processamento de Dados da Uneb, no seu projeto final de conclusão do curso, onde fui o orientador do trabalho. A construção foi iniciada pela construção das diversas classes que compõem a camada de persistência.

A classe PersistentConnection que tem a principal função de suportar a conexão aos diversos tipos de bancos de dados relacionais. As propriedades do

banco de dados para o qual será feita a conexão, serão passadas via arquivo de configuração, como parâmetro no momento da execução. O principal método da classe PersistentConnection é o método construtor que tem a função de criar uma conexão JDBC para um banco de dados especificado. Os parâmetros de entrada são: *driver, url, username, password, name*, onde *name* é o nome do banco de dados.

Após a criação da conexão, o próximo passo é o gerenciamento da conexão, que é a função da classe ConnectionBroker. Durante a inicialização da camada de persistência o método init() é chamado e então de posse da conexão criada por PersistentConection, armazena o nome do banco de dados para o qual foi feita a conexão, bem como as suas propriedades em lista de bancos de dados que é representada pela variável database_list. Os dados sobre o banco de dados são trazidos através de um *select* em uma tabela do banco de dados que contém, em cada coluna um atributo do banco de dados. O nome da tabela do banco de dados é fornecido pelo arquivo de configuração.

Depois de resolvidas as principais necessidades de conexão, o próximo passo é focar o objeto persistido. Neste caso, como trazer o objeto de volta de um banco de dados relacional. A classe PersistentObject encapsula o comportamento necessário para poder criar instâncias de objetos persistentes, podendo assim, um objeto persistente ser salvo, recuperado, atualizado ou excluído de um mecanismo persistente. Ela sabe como armazenar seus atributos de uma linha de resultados de uma tabela do banco de dados. O método chave da classe PersistentObject é o método setAttributes que tem a função de guardar os atributos de objetos a partir de dados recuperados de uma tabela do banco de dados. Para a completa e correta execução deste método, a classe PersistentObject precisa se apoiar nas classes de mapeamento, que são as classes DataDictionary e ClassDictionary.

A ClassDictionary encapsula o comportamento necessário para mapear classes e atributos em tabelas e colunas em um banco de dados relacional. O construtor da classe é o método chave da classe ClassDictionary. É responsável por carregar um dicionário de dados (Data_Dictionary) para uma classe específica e pegar o nome da tabela para uma dada classe, que é passada como parâmetro. Os atributos class_map, reverse_class_map armazenam respectivamente o mapeamento entre atributos-colunas e colunas-atributos. O atributo primary_keys é um vetor e tem a função de armazenar as chaves primárias de cada tabela no banco

de dados. O atributo dictionary tem a função de armazenar o nome da classe do objeto persistente. O método ClassDictionary utiliza PersistentConnection para criar uma conexão para o banco de dados. Este método faz um select na tabela Data_Dictionary para buscar o nome da tabela correspondente ao nome da classe e então carrega as tabelas de atributos de mapeamento através do método loadTableAttributes().

Agora que as classes de conexão ao banco e as classes de mapeamento estão criadas, pode-se voltar para o desenvolvimento da classe PersistentObject que encapsula o comportamento necessário para poder criar instâncias de objetos persistentes. Para armazenar atributos de objetos de uma linha de resultados de uma tabela do banco de dados a classe PersistentObject utiliza o método setAttributes(). Este método recebe como parâmetros o conjunto resultado de uma consulta a uma tabela do banco de dados, o dicionário de dados para aquele nome de tabela e um metadado associado às colunas retornadas.

Agora de posse das classes de conexão ao banco, das classes de mapeamento e da classe que trata do objeto em questão, que é a classe PersistentObject, tem-se agora que implementar as classes que encapsulam mecanismos pelos quais se possa criar expressões SQL de *SELECT, INSERT, UPDATE e DELETE*. A hierarquia de classes SQLStatement tem essa função. A classe SQLStatement é uma classe abstrata e será a super-classe das classes SQLSelectStatement, SQLDeleteStatement, SQLInsertStatement e SQLUpdateStatement.

Para o melhor entendimento dos métodos desta hierarquia de classes, começamos pelo estudo dos métodos que as classes filhas implementam e depois, partir para os métodos que as classes filhas utilizam importando da classe mãe, ou seja, os métodos que são implementados na classe mãe.

A classe PersistentSource (que é a classe fonte de funções delete() , save() e retrieve() para o usuário) é a classe que mais utiliza os recursos desta hierarquia de classes. Dentro de uma operação de exclusão inicialmente instancia-se a classe SQLDeleteStatement(), que cria o início da expressão SQL pelo uso do dicionário de classes que é passado como parâmetro.

Para executar uma operação sobre um subconjunto de objetos do banco de dados deve-se estabelecer um critério na operação. A classe PersistentCriteria tem a principal função de gerenciar todas as classes que implementam algum critério de

seleção. As classes que implementam algum critério de seleção estão implementadas em um esquema de hierarquia de classes. A classe abstrata se chama SelectionCriteria e é a classe mãe das classes LessThanCriteria, LikeCriteria, GreaterThanCriteria, EqualToCriteria.

As classes Criteria são de livre acesso para o desenvolvedor de aplicações. A definição de um critério começa instanciando a classe PersistentCriteria, que possui o método construtor que através do atributo criteria (que é do tipo Vetor), armazena a coluna e o valor da coluna a ser utilizada como critério.

A classe PersistentSet encapsula as básicas características de um cursor de banco de dados. Cursores permitem recuperar um subconjunto de informações de uma só vez. Isto tudo acontece instanciando-se a classe PersistentSet a partir da classe PersistentSource (que é a principal classe de acesso para o desenvolvedor) que chama o seu método construtor8.7 Classe que permite a execução de um conjunto de operações de uma só vez.

A classe Trans (transação) permite um conjunto de operações em objetos para ser agrupadas em uma simples operação que pode ser bem sucedida ou falhar como um todo. Para isso o desenvolvedor deve instanciar a classe Trans. O método construtor tem a função de inicializar a classe Connection_broker que por sua vez verifica se existe um banco na lista de banco de dados (database_list). O método também instancia a classe DataDictionary.

A classe PersistentSource bem como as classes Trans e PersistentCriteria são classes de acesso para o desenvolvedor, isto é, são classes onde o desenvolvedor busca métodos de forma a fazer objetos persistirem em um banco de dados relacional.

No caso da classe PersistentSource, todos os seus métodos são de interesse do desenvolvedor. Têm-se diferentes tipos de métodos para uma mesma operação sobre o banco de dados. Por exemplo, a operação de excluir um objeto de um banco de dados, pode ser feita usando ou não um critério de exclusão e fazendo ou não parte de uma transação. Também existe uma diferença entre um método que é chamado pelo desenvolvedor daquele que é chamado pela classe Trans(transação). O método que é chamado pelo desenvolvedor tem que criar uma conexão persistente, isto já não é preciso no método que é chamado pela classe Trans, pois a classe Trans já possui uma conexão persistente para aquela operação.

4.7 Testes

Para se testar a camada de persistência foi necessária a construção de uma aplicação utilizando a linguagem Java. O programa “exemplo” implementou um sistema simples de compra de carros. Consta de uma lista de carros cadastrados, uma lista de clientes cadastrados e uma lista de ordens de compra associando um cliente ao(s) carro(s).

O sistema é composto por quatro classes: Exemplo, Cliente, Carro e Ordem. A classe Exemplo é responsável pelo gerenciamento do sistema, cabendo a ela o gerenciamento do menu e dos sub-menus e implementa os métodos que criam operações no sistema utilizando os métodos da camada de persistência. Também é responsável por receber um arquivo de configuração para conexão ao banco de dados.

Esta aplicação teste também foi construída pelos alunos referidos no item anterior e demonstrada a sua utilização, fazendo uso da camada de persistência.

O sistema é dividido por menus e sub-menus de operação que são gerenciados pela classe Exemplo.

Na entrada do sistema o seguinte menu principal aparece:

```
**************************************
Menu principal do programa de teste
**************************************
1. Menu de operações para o Cliente
2. Menu de operações para o Carro
3. Menu de operações para a Ordem
0. Sair
Escolha a sua opção:
```

O usuário tem a opção de escolher um dos três sub-menus de operações, que são descritos a seguir:

```
**************************************
Menu Cliente
**************************************
```

1. Criar um novo cliente

2. Listar clientes

3. Mudar nome do cliente

4. Excluir cliente

5. Encontrar cliente

9. Retornar ao menu inicial

0. Sair

Escolha uma opção:

****

Menu Carro

****

1. Criar um novo carro

2. Listar carros

9. Retornar ao menu principal

0. Sair

Escolha uma opção:

****

Menu Ordem

****

1. Criar nova ordem

2. Listar ordens

9. Retornar para o menu principal

0. Sair

Escolha uma opção:

Como explicitado acima, a classe Exemplo é responsável por receber as propriedades da conexão ao banco e passar como parâmetros para o método construtor da classe PersistentConnection da camada de persistência, que é responsável por criar uma conexão persistente. Um arquivo de configuração é usado para abrigar as configurações que são usadas para conectar ao bando de dados Oracle. Para o teste da camada de persistência, foi utilizado o Personal Oracle. O arquivo de configuração usado neste exemplo é descrito a seguir.

driver=sun.jdbc.odbc.JdbcOdbcDriver
url=jdbc:odbc:Oracle
username=scott
password=tiger
data_dictionary=data_Dictionary
database_table=databases

No arquivo de configuração, a linha *data_dictionary* passa para a classe Exemplo o nome do dicionário de dados que foi criado no banco de dados. A linha *database_table* passa para a classe Exemplo o nome da tabela no banco de dados que armazena os dados sobre as configurações utilizadas para se criar uma conexão persistente.

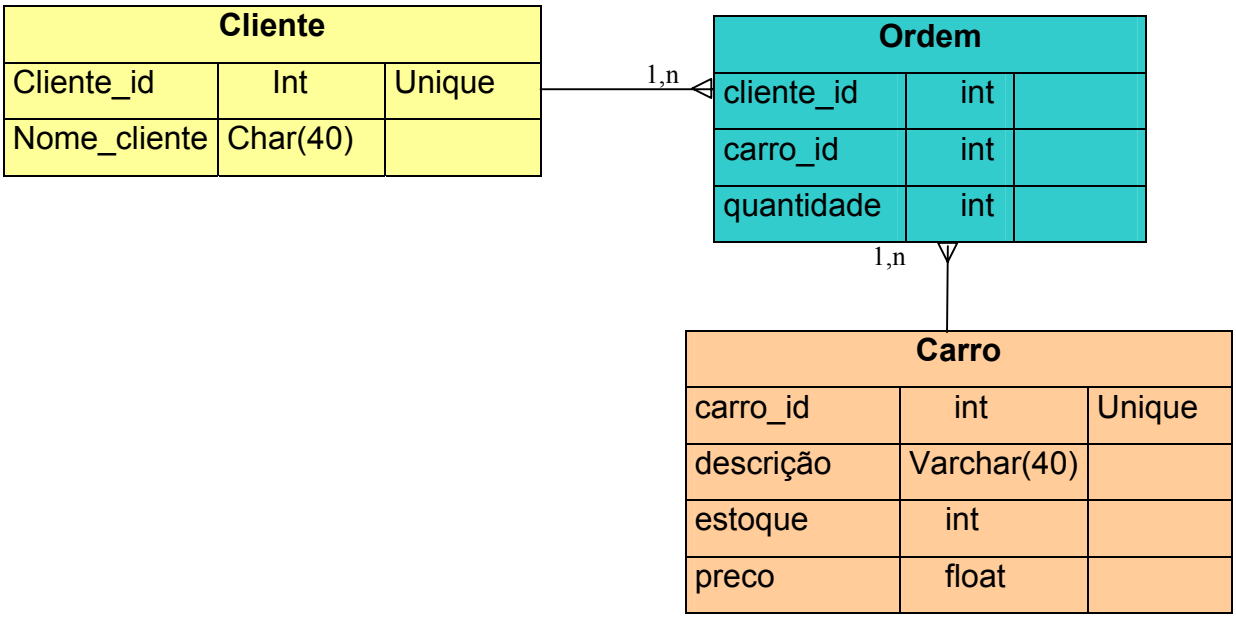
Antes de executar o programa exemplo é necessário criar algumas tabelas no banco de dados que são responsáveis por armazenar o seguinte:

- As informações sobre a conexão persistente (tabela databases).
- As informações sobre o mapeamento entre nome de classes e os respectivos nomes de tabelas (tabela data_dictionary).
- As informações sobre o mapeamento entre atributos de uma classe e respectivas colunas de uma tabela (tabela class_dictionary).

Além da criação dessas tabelas, tem-se que inserir os dados necessários para que a aplicação exemplo utilize a camada de persistência. Na hora de inserir as linhas na tabela class_dictionary é que se definem quais são os atributos que devem ser persistidos. Só serão inseridas linhas na tabela class_dictionary para os atributos que devem ser persistidos.

Também é necessário criar as tabelas para as classes de negócio utilizadas no programa exemplo, que são as classes Carro, Cliente e Ordem. A estratégia utilizada nesta camada de persistência é a criação de uma tabela por classe concreta. O esquema de tabelas criadas para as correspondentes classes de negócio é mostrado na Figura 65.

Figura 65: Esquema de tabelas criadas no banco para o programa Exemplo.



A seguir é mostrado um *script* que foi criado para realizar todas essas operações descritas anteriormente:

```
; exemplo.sql
DROP TABLE databases;
DROP TABLE data_dictionary;
DROP TABLE class_dictionary;
; cria a tabela que armazenará os dados sobre a conexão ao banco de dados
CREATE TABLE databases (
        database        char(20)        PRIMARY KEY,
        driver          varchar(100),
        locatorvarchar(100),
        username        char(8),
        passwdchar(8),
        connections     integer,
        vendor char(20)
        );
; cria a tabela que armazenará o mapeamento entre classes e tabelas.
CREATE TABLE data_dictionary (
```

```
        class_name    char(128)       PRIMARY KEY,
        database      char(10),
        table_name    char(40)
);
; cria a tabela que armazenará o mapeamento entre atributos e colunas.
CREATE TABLE class_dictionary (
        table_name    char(40),
        class_attribute varchar(64),
        table_column  varchar(64),
        is_primary    char(1)
);
;insere os dados sobre a conexão ao banco de dados
INSERT INTO databases VALUES ('primary', 'sun.jdbc.odbc.JdbcOdbcDriver',
'jdbc:odbc:Oracle', 'scott', 'tiger', -1, 'Oracle');
; insere o mapeamento entre a classe Cliente  e a tabela cliente.
INSERT INTO data_dictionary VALUES ('exemplo.Cliente', 'primary', 'cliente');
; insere os mapeamentos entre atributos e colunas para a tabela referente a classe
; do atributo, apenas serão  criadas linhas para os atributos que devem persistir.
INSERT INTO class_dictionary VALUES ('cliente', 'cliente_id', 'cliente_id', 'y');
INSERT INTO class_dictionary VALUES ('cliente', 'nome_cliente', 'nome_cliente', 'n');
DROP TABLE cliente;
; cria-se a tabela  cliente para uma classe concreta Cliente
CREATE TABLE cliente (
        cliente_id int UNIQUE,
        nome_cliente  char(40)
);
; insere o mapeamento entre a classe Carro  e a tabela carro.
INSERT INTO data_dictionary VALUES ('exemplo.Carro', 'primary', 'carro');
;insere os mapeamentos entre atributos e colunas para a tabela referente a classe
; do atributo, apenas serão criadas linhas para os atributos que devem persistir.
INSERT INTO class_dictionary VALUES ('carro', 'carro_id', 'carro_id', 'y');
INSERT INTO class_dictionary VALUES ('carro', 'descricao', 'descricao', 'n');
INSERT INTO class_dictionary VALUES ('carro', 'estoque', 'estoque', 'n');
INSERT INTO class_dictionary VALUES ('carro', 'preco', 'preco', 'n');
```

```
DROP TABLE carro;
; cria-se a tabela  carro para uma classe concreta Carro
CREATE TABLE carro (
   carro_id int UNIQUE,
   descricao varchar(40),
   estoque int,
   preco float
);
; insere o mapeamento entre a classe Ordem  e a tabela ordem.
INSERT INTO data_dictionary VALUES ('exemplo.Ordem', 'primary', 'ordem');
;insere os mapeamentos entre atributos e colunas para a tabela referente a classe
; do atributo, apenas serão criadas linhas para os atributos que devem persistir.
INSERT INTO class_dictionary VALUES ('ordem', 'carro_id', 'carro_id', 'y');
INSERT INTO class_dictionary VALUES ('ordem', 'cliente_id', 'cliente_id', 'y');
INSERT INTO class_dictionary VALUES ('ordem', 'quantidade', 'quantidade', 'n');
DROP TABLE ordem;
; cria-se a tabela  carro para uma classe concreta Carro
CREATE TABLE ordem (
   carro_id int,
   cliente_id int,
   quantidade int
);
commit;
```

A classe Exemplo vai testar os métodos responsáveis por fazer objetos persistirem em um banco de dados relacional.

4.8 Avaliação do Estudo de Caso

Neste projeto foi implementada uma camada de persistência utilizando um banco de dados relacional como repositório de dados, tendo sido feita uma modelagem utilizando o padrão UML e uma implementação utilizando a linguagem de programação Java.

A principal dificuldade encontrada durante a implementação foi a necessidade de se evitar ao máximo a geração de código rígido (*hard-coded*) SQL, que é a criação de expressões de consultas, inserção ou exclusão com nomes de tabela e colunas estáticos dentro do código fonte da camada de persistência. Para evitar este problema, foi necessário implementar maneiras dinâmicas de se obter os nomes de tabelas e colunas a serem consultadas no banco de dados. Porém isto foi necessário, já que mudanças simples poderiam ser feitas no banco de dados, onde tabelas e/ou colunas poderiam ser movidas ou renomeadas. Se fosse utilizado *hard-coded*, então estas alterações causariam mudanças no código da camada de persistência.

A estratégia de mapear herança utilizando uma tabela por classe concreta, provou ser uma maneira eficaz, que obtém os objetivos desejados, mesmo sendo a estratégia mais fácil de implementar. Esta estratégia apresenta uma facilidade de operação por parte do desenvolvedor, pois ele precisa simplesmente criar as tabelas correspondentes às classes concretas existentes em sua camada de domínio e preencher as linhas das tabelas de dicionário de classes e dicionário de dados no banco de dados.

A idéia inicial da construção de uma camada de persistência era que o desenvolvedor não precisasse ter o mínimo conhecimento de como os objetos são salvos no mecanismo persistente, ele precisaria apenas invocar comandos tais como *save, retrieve e delete* para a camada de persistência, mas isso não foi totalmente alcançado: o desenvolvedor tem que invocar também comandos das classes que implementam cursores em banco de dados e critérios de seleção ou exclusão; os comandos da classe de transação, para iniciar uma transação e entregar uma transação e os comandos de inicialização de uma conexão persistente.

A ação de tirar da camada de persistência as informações sobre as classes do domínio de negócio, bem como as informações sobre a conexão ao banco de dados, tem a vantagem de aumentar a reusabilidade e modularidade da camada e, em muitos casos, favorecer o desempenho e a manutenção da aplicação que a utiliza.

5 CONCLUSÕES

A utilização de componentes de software está revolucionando a indústria. As empresas que decidiram integrar a tecnologia de componentes com a sua infra-estrutura de tecnologia da informação estão enfrentando grandes desafios técnicos e organizacionais.

Porque então as empresas devem investir em projetos baseados na utilização de componentes? Que ganhos de produtividade e qualidade podemos esperar da utilização de uma metodologia de desenvolvimento de software utilizando a tecnologia de componentes? Que aspectos técnicos influenciarão os desenvolvedores, que lhes garantam que os investimentos realizados em treinamento e capacitação de recursos humanos serão plenamente recompensados? Que mudanças organizacionais serão necessárias para implantação de uma metodologia de desenvolvimento de software baseada em componentes?

Simplesmente porque os benefícios do desenvolvimento bem gerenciado, baseado em componentes, já foram documentados e comprovados em várias experiências. O desenvolvimento rápido proporcionado pelo reuso de componentes, testados e adequados, combinado com a facilidade de manutenção e escalabilidade, garantem uma diminuição de tempo e esforços ao longo dos anos. Os ganhos em qualidade e produtividade que foram conseguidos justificam o alto custo da introdução dessa nova tecnologia, considerando o custo de consultores para repassar a experiência, a aquisição de novas ferramentas, técnicas, processos, hardware e a não menos famosa "curva de aprendizado".

As características desejáveis para projetos bem sucedidos incluem: sistemas corporativos de grande porte com um potencial grande de reusabilidade de componentes, perspectiva de vida útil longa, grande probabilidade de modificações e extensões e pessoas interessadas em trabalhar em projetos desta natureza.

Projetos mal sucedidos tipicamente incluem esforços ambiciosos com expectativas e planejamento irreais, falta de conhecimento adequado e falhas na organização.

A escolha de um projeto piloto para internalização e consolidação da tecnologia deve ser a forma de testar a metodologia de desenvolvimento adotada e

treinar os técnicos da organização. Esses profissionais devem sofrer uma “imersão total” nas novas tecnologias e metodologias, para saber o que fazer, e não utilizarem a nova tecnologia com os conhecimentos que têm, fazendo portanto uma utilização errônea ou, no mínimo, uma subutilização dos recursos disponíveis.

Finalmente, o enfoque de transferência de tecnologia como um processo de educação de pessoas é o único que pode garantir o sucesso de uma transição tecnológica. A compra de hardware e software de última geração não garante em absoluto a melhoria dos sistemas. Desenvolvedores capazes, bem preparados e bem suportados por ferramentas adequadas são a única garantia de sucesso no desenvolvimento.

Na visão de uma fábrica de software, o complexo processo de desenvolvimento de componentes é dividido em fases simplificadas, com entradas e resultados bem-definidos. Dessa forma, é evidente o uso de especialização profissional em fases, processos de negócios e ferramentas, onde as especialidades de construção de componentes são especificadas.

A missão de uma fábrica de software é maximizar a produção de software, através da melhoria contínua do processo de desenvolvimento, com a utilização de metodologias, técnicas e ferramentas que direcionem os esforços de toda a equipe. Nesse sentido é imprescindível a utilização de padrões para o eficaz funcionamento de seus processos.

Outra característica que uma fabrica de software é busca de soluções definidas por um conjunto de elementos interdependentes, com funcionalidades e interfaces padronizadas, buscando-se dessa forma a minimização da complexidade da solução proposta, pela utilização de componentes de software. Esse processo incremental de reutilização de componentes é impulsionado à medida que cresce a demanda pela fábrica, aumenta a produção e o acervo de componentes, realimentando a diminuição de custos.

A área de desenvolvimento de software tem se caracterizado pela criação de soluções especificas para cada situação. Obviamente, fica evidente que construir sistemas é uma atividade onerosa para as empresas. Na maioria das vezes, depois que se consegue definir os requisitos essenciais da aplicação, passando pelas fases de análise, projeto e construção, o cliente já mudou ou incorporou novas funcionalidades a sua necessidade. Enquanto as soluções personalizadas são

demoradas e caras, as soluções genéricas - pacotes prontos – consomem recursos na adaptação do cliente.

Assim, como aconteceu com outras ciências, a solução para este problema passa obrigatoriamente pela modernização dos meios de produção, com a utilização das tecnologias de componentes no desenvolvimento de sistemas.

5.1 Recomendações para Trabalhos Futuros

Neste trabalho analisamos os aspectos metodológicos e gerenciais da utilização da tecnologia de desenvolvimento de sistemas baseado em componentes. No entanto, muitos outros aspectos técnicos devem ser analisados para termos o domínio da tecnologia de componentes. Alguns aspectos abaixo descritos devem ser objeto de estudos futuros, tal a sua importância para a tecnologia de componentes.

Em relação aos sistemas legados, principalmente os sistemas corporativos desenvolvidos para mainframes, que utilizam tecnologia e arquitetura proprietários e que já estão consolidados na organização. Como fazer com que estes aplicativos sejam incorporados em novas aplicações, encapsulados em componentes de software e utilizados de forma independente de linguagens de programação, plataforma de banco de dados e ambiente operacional?

Como tratar o versionamento de componentes? Em um ambiente de desenvolvimento de aplicações baseado em componentes, dezenas e até mesmo centenas de componentes são criados e utilizados em diversas aplicações. Posteriormente, estes componentes podem ser atualizados, gerando-se novas versões de um mesmo componente. Como distribuir estes novos componentes entre as diversas aplicações usuárias?

Componentes de software podem ser adquiridos de fabricantes idôneos ou podem ser desenvolvidos internamente na própria organização. Decidir quanto à adoção de componentes adquiridos ou não envolve ponderar acerca de diversos aspectos, tais como a reputação dos fabricantes de componentes.

Um componente de software tem uma ou mais interfaces externas distintas de sua implementação e são definidas de maneira contratual entre as partes. Como são as características destas interfaces para permitir a utilização dos componentes em

um ambiente de aplicações distribuídas, conforme citado no item 2.4, páginas 16-17, utilizando os padrões CORBA, DCOM ou EJB?

REFERÊNCIAS

AMBLER, Scott W. Análise e Projeto Orientados a Objetos Volume 2. Rio de Janeiro: Infobook, 1999.

BOOCH, Grady; RUMBAUGHT, James; JACOBSON, Ivar. ***The Unified Software Development Process***.

BOOCH, Grady; RUMBAUGHT, James; JACOBSON, Ivar. **UML Guia do Usuário**. São Paulo: Campus, 2000.

BROWN, Alan W. **Large-Scale, Component-Based Development**. USA: Prentice Hall PTR, 2000.

D'SOUZA, Desmond F.; WILLS, Alan C. **Objects, Components, and Frameworks with UML: The Catalysis Approach.** USA: Addison Wesley Longman, 1999.

FURLAN, José Davi. **Modelagem de Objetos Através da UML.** São Paulo: Makron Books, 1998.

JACOBSON, Ivar. **The Object Advantage: Business Process Reengineering with Object Technology**. USA: Addison-Wesley, 1995

JACOBSON, Ivar; GRISS, M.; JOHNSSON, P. **Software Reuse: Architecture, Process and Organization for Business Success**. USA: Addison-Wesley Longman, 1998.

KHOSHAFIAN, Setrag. **Banco de Dados Orientado a Objeto.** Rio de Janeiro: Infobook, 1994.

KIRTLAND, Mary. **Projetando Soluções baseadas em Componentes.** Rio de Janeiro: Campus, 1999.

KRUCHTEN, Philippe. **The Rational Unified Process An Introduction Second Edition.** USA: Addison Wesley, 2000.

LARMAN, Craig. **Utilizando UML e Padrões**, Rio de Janeiro: Campus, 2000

LARSON, L.A. **Database Directions: from Relational to Distributed, Multimedia, and Object-Oriented Database Systems**. USA: Prentice Hall PTR, 1995

MORISSEAU-LEROY, Nirva; SOLOMON, Martin K.; BASU, Julie. **Oracle 8i Programação de Componentes Java com EJB, CORBA e JSP**. Rio de Janeiro: Campus, 2001.

ORFALI, R. e HARKEY, D. **Client/Server Programming with JAVA and CORBA**. USA: Wiley Computer Publishing, 1997

QUATRANI, Terry. **Visual Modeling with Rational Rose 2000 and UML**. USA:

Addison Wesley Longman, 2000.

SOLOMON, Martin; MORISSEAU-LEROY, Nirva; MOMPLAISIR, Gerald. **Oracle8i SQL Programming**. . Rio de Janeiro: Campus, 2000.

SZYPERSKY, Clemens. **Component Software**. USA: Addison-Wesley, 1998.

SZYPERSKY, Clemens. **Component Software: Belond Object – Oriented Programming**. USA: Addison-Wesley, 1997.

TINDALL, Paul. **Desenvolvendo Aplicações Corporativas com VB, MTS, IIS, SQL Server e XML**. Rio de Janeiro: Campus, 2000

Ambler, Scott W. Design a search Screen. **Software Developing**, Janeiro 1997, p79-82.

Ambler, Scott W. Mapping Objects to Relational Databases. **An AmbySoft Inc. White Paper**. Fevereiro 1998.

Ambler, Scott W. Mapping Objects to Relational Databases. Software Development, Outubro 1995, p.63-72.

Ambler, Scott W. The Design of a Robust Persistence Layer For Relational Databases. **An AmbySoft Inc. White Paper**. Outubro 1997-1999.

Ambler, Scott W. The realities of Mapping Objects to Relational Databases. **Software Developing**, Outubro 1997, p.71-74.

BELLOQUIM, Átila. Migração Tecnológica É um Processo Cultural. **Developers' Magazine.** Rio de Janeiro: Axcel Books, Janeiro 1997.

BELLOQUIM, Átila. Reutilização de Objetos: Promessa ou Dívida?. **Developers' Magazine.** Rio de Janeiro: Axcel Books, v.5, n.53, p.10-11, Janeiro 2001.

BRAGA, Regina M.M. Um Passo à Frente na OO: Desenvolvimento por Componentes. **Developers' Magazine.** Rio de Janeiro: Axcel Books,v.3, n.26, p.18-21, Outubro 1998.

Metodologia VINCIT. Rio de Janeiro: Microservice. Fuzion Engenharia de Software Ltda, [1998]. 1 CD-ROM. Windows 3.1/98.

Organizing for Components. **Component Strategies**. Fevereiro 1999.

RIDOLFI, Lorenzo. Sistema Cliente / Servidor Baseados em Componentes. **Developers' Magazine.** Rio de Janeiro: Axcel Books, v.3, n.26, p.12-14, Outubro 1998.

SANT'ANNA, Mauro. .Net Framework: Programação Baseada em Componentes. **Developers' Magazine.** Rio de Janeiro: Axcel Books, Fevereiro 2001.

CATALYSIS – Metodologia – Disponível em http://www.iconcomp/catalysis

Component Software Glossary – glossário de termos técnicos da área de componentes. Disponível http://www.objs.com/survey/ComponentwareGlossary.htm

Documentação sobre CORBA. http://adams.patriot.net/~tvalesky/freecorba.html

Object Managment Group uma entidade independente que normatiza as aplicações orientadas a objeto. Lá também se encontra a documentação oficial da UML, artigos sobre CORBA e outros projetos – Disponível em http://www.omg.org

Oracle Corporation. Documentação on-line. Disponível em http://www.oracle.com

PAU no Gato! Por que? Rio de Janeiro: Sony Music Book Case Multimídia Educacional, [1990]. 1 CD-ROM. Windows 3.1.

Rational onde foi desenvolvida a UML e que possui diversos artigos dos autores da notação e ferramentas de software de apoio à notação – Disponível em http://www.rational.com.br

Revista Software Development com diversos artigos nacionais sobre UML e outros assuntos ligados ao desenvolvimento de software – Disponível em http://www.sdmagazine.com/uml/home.htm

SAKAMOTO, Ângela R. O RUP e a Choose Technologies Artigo publicado na página da Choose Technologies. Disponível em http://www.choose.com .

Sun Microsystems. Documentação sobre Java. Disponível em http://www.java.sun.com

Tutorial sobre CORBA. http://www.cs.indiana.edu/hyplan/kksiazek/tuto.html

Website de busca com mais de 9000 links sobre orientação a objetos e busca de assuntos ligados à tecnologia de objetos e programação – Disponível em http://www.cetus-links.org/